Contributions to Software Runtime for Clustered Manycores Applied to Embedded and High-Performance Applications

Julien Hascoët

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

INSA RENNES

THESE DE DOCTORAT DE

L'INSA RENNES
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N°601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Signal, Image, Vision

Par

Julien Hascoët

Contributions to Software Runtime for Clustered Manycores Applied to Embedded and High-Performance Applications

Thèse présentée et soutenue à Rennes, le 14/12/2018
Unité de recherche : IETR
Thèse N° :

Rapporteurs avant soutenance :

François Irigoin
Professeur à MINES ParisTech
Alain Girault
Directeur de Recherche à l'INRIA Grenoble

Composition du Jury :

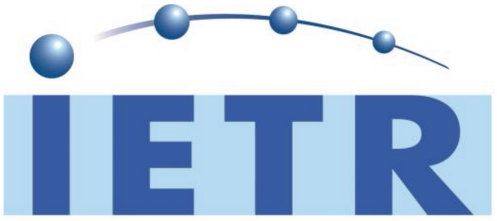
Alix Munier-Kordon
Professeur au LIP6 Paris / Présidente du jury
François Irigoin
Professeur à MINES ParisTech / Rapporteur
Alain Girault
Directeur de Recherche à l'INRIA Grenoble / Rapporteur
Jean-François Nézan
Professeur à l'INSA de Rennes / Directeur de thèse académique
Benoît Dupont de Dinechin
Directeur Technique à Kalray / Directeur de thèse entreprise
Karol Desnos
Enseignant Chercheur à l'INSA de Rennes / Encadrant de thèse

Invité :

Jeronimo Castrillon
Professeur à l'Université Technique de Dresden

Contributions to Software Runtime for Clustered Manycores Applied to Embedded and High-Performance Applications

Julien Hascoët



Contents

Acknowledgments

To my family, my love ones, my friends, and my colleagues.

I express my sincerest thanks to my supervisors for their advice and time, Jean-François Nézan, Benoît Dupont de Dinechin, and Karol Desnos.

I wish to thank Alain Girault and François Irigoin for their hard work on reviewing this manuscript.

I thank all my work colleagues for their technical support on the software side: Pierre Guironnet de Massas, Benoît Dupont de Dinechin, Samuel Jones, Patrice Gerin (special thanks to the simulator without which I would be still debugging my first contribution), Frederic Blanc, Karol Desnos, Minh Quan Ho, Marius Gligor, Julien Villette; and on the hardware side: Vincent Ray, Nicolas Brunie, and Alexandre Blampey.

CHAPTER 1

Introduction

1.1 General Context

Computer science is a wide area, and it has changed our ways of living. From the invention of the transistor in the fifties to the latest released computer games, self-driving cars, and smart homes, the world as we know is now dependent on computer systems. Ever since the emergence of computer science and computer architecture, the need to compute has always been growing. The computing requirements of applications increase as the services provided by them make our lives easier, smarter and faster in diverse ways.

In this Ph.D., the focus is on embedded systems. In general, an embedded system is everything that is not a general-purpose computer in the computer field. Embedded systems are found in all electronic systems or parts of the electronic systems. These new intelligent systems are found everywhere. For instance, typical embedded systems are smartphones, fitness wristband, Global Positioning Systems (GPSs), MP3 digital audio players, dishwashers, Television (TV) systems and almost everything powered by electricity requiring intelligence to provide a human being with a specific service. Today, thanks to the high-density integration of transistors (5 nanometers are announced by 2020), embedded systems can execute many complex tasks and with high energy efficiency. Thus, embedded systems are now everywhere and unavoidable, and they have a significant impact on our lives.

Embedded systems are found in several new application domains like the Internet of Things (IoT), big data, Advanced Driver-Assistance System (ADAS) [MGL+16] and embedded vision and video. Recent examples of complex embedded computing systems can also be found in the aerospace area where embedded electronic devices must meet high expectations regarding fault tolerance, robustness, redundancy and system isolation for application safety. Companies like Airbus [air18] spend years designing, implementing and testing in real-life conditions embedded systems before putting them into service. This is due to the application specificities, especially in complex and constraint environments.

As said in paper [Wol02], *"an embedded system is a computer system with dedicated functions within a larger mechanical or electrical system, often with real-time constraints."* Therefore, an embedded system is designed to fit a specific application in a constrained environment. Constraints are usually both hardware and software. For instance, constraints of embedded systems are generally the power consumption, timing response latency, time

computing determinism, computing performance, robustness, cost, area and fault tolerance. Some of the listed constraints are antagonists. Indeed optimizing the latency of an application usually ends up in reducing its throughput. Or even, increasing the performance of an application leads to increasing the power consumption of the final system, as it goes faster; therefore, it consumes more energy. Finally, increasing performance often means increasing the cost of the application. Embedded systems are known to be complicated to design, implement and verify that they satisfy the required features with the related constraints. That is why the hardware needs to be leveraged by the software to make the application run efficiently in a constraint environment.

In 1965, a now-famous paper by Gordon Moore observed that every two years the number of transistors in a densely integrated circuit doubles. In addition to this Moore's law, computer manufacturers benefited from the Dennard scaling of MOSFET technology, which states that as transistors get smaller and faster, their power density stays constant so that the power use stays in proportion with the area. As a result, manufacturers continuously increased the operating frequency of integrated processors from the old Intel® 4004® processing clock gated at 740 kHz in 1971 to the 2004 high-end Intel® Pentium®-4 clock gated at 4 GHz. The Dennard scaling broke down around 2004, corresponding to the 65nm CMOS technology nodes. Since then, semiconductor manufacturers have lost the ability to increase clock frequencies significantly without hitting a power wall. Since 2004, computer architects design parallel architectures to improve the computational capabilities without hitting this power wall.

Before 2004, computer architects already designed some parallel architectures. An example of them is the Texas Instruments (TI) Multiprocessor System-on-Chip (MPSoC), called C80 [DS95], which integrates 4 Digital Signal Processor (DSP) cores with a Reduced Instruction Set Computer (RISC) core, designed to target energy-efficient computing for embedded systems. But this MPSoC could only be programmed by TI's experts, showing that multi-core architectures are more challenging to use. Multi-core software is indeed very different from single-core software due to concurrent executions, the sharing of resources and many other issues that will be discussed in this Ph.D. dissertation.

Increasing the number of cores using shared memory enables better performances up to a certain point (around up to tens of cores). At this point, the performances are limited by concurrent accesses to a single memory resource (shared memory). Nowadays, a new generation of architectures has been designed, called clustered architectures. These new architectures feature distributed memories to tackle the problem of shared memory. These new architectures are even more complicated to be programmed, involving huge software development costs.

The memory bandwidth is a significant performance factor. Most of the time, the main memory bandwidth (bandwidth between a processor and an external memory) is the bottleneck of high-performance applications onto parallel architectures. The main memory bandwidth is lower compared to the available internal on-chip memory bandwidth [WWP09]. When the number of cores is increasing inside a processor, memory bandwidths often become the bottleneck of the application. In this case, the cores are not fed with data as fast as they can process them. The purpose of the memory hierarchy is to reduce the average memory latency access time and to reduce the memory traffic between memories. Avoiding the memory bandwidth wall requires temporal and spatial data locality to use the on-chip memory efficiently. It is crucial to handle such a bottleneck early in optimization stages, to get competitive performances on this new generation of clustered manycores.

In this context, this Ph.D. aims at reducing the complexity of the software development to decrease the development time, while keeping performance, onto these new generations of complex MPSoCs.

1.2 Scope of the Thesis

This thesis is a collaboration between the Institute of Electronics and Telecommunications of Rennes (IETR)'s Video Analysis and Architecture Design for Embedded Resources (VAADER) team and the Kalray French semiconductor company.

The focus is on a family of manycore architectures designed by the Kalray company. This family of manycores features distributed memories to let parallel programs scale. Kalray's manycore architectures regroup cores into clusters that are fitted with a multi-banked shared local memory; and then, these clusters are interconnected with a network-on-chip. This MPSoC enables the use of up to 288 real cores running in parallel with their own execution context. MPPA® is built for energy-efficient computing using the local memories and Direct Memory Access (DMA) engines to move data off-chip on on-chip by software. Such characteristics make the MPPA® processor a severe competitor regarding low-power computing, but it is challenging to program, observe, and debug.

We study in this thesis programming models and Application Programming Interfaces (APIs) for the latest generation of clustered architectures. Both programming models and APIs abstract the targeted architecture using high-level models and documented functions.

APIs provide a set of functions enabling one or several features to be run onto a processor. APIs are found at several levels of programming. Low-level programming is very close to the targeted hardware. Programming at low-level is painful and requires an in-deep knowledge of the targeted architecture. The portability of a low-level program is also very poor. But it comes with unmatched performances when the programming is properly performed. High-level and mid-level programming abstract hardware architectures to the programmer. Programmers use high-level APIs, and the program can be compiled and executed on several chips including usually general purpose processors and a given set of processors for embedded systems. However, high-level APIs can also be found with high-performance optimized libraries provided and sold by hardware manufacturers that most of the time unleash the best performance for the use-cases addressed by the API. See for instance the Intel® Math Kernel Library (MKL). The performances of those API comes with a loss in terms genericity for the programmer.

A programming model aims to be a way or a style to express a computation. A programming model is usually language agnostic whereas an API is not. An example of programming model is OpenMP that is expressed using a specific API. The application engineer only writes few isolated compilation directives in its source code, and if supported, the sequential computation is automatically executed in parallel according to the directives.

In this thesis, the aim is to use dataflow programming models to describe an application at a high level of abstraction. Many dataflow models exist, and they are widely used in software compilers that are either low-level or high-level. Dataflow applications are represented with computing blocks and exchange data using directed communication links. As such a dataflow is usually represented as a directed graph where nodes represent the computation and links between nodes represent the data communications. However, the dataflow programming models have various refinements when going into details. It might sometimes be troublesome for application engineers when complex applications have to be ported efficiently. But still, dataflow programming is a great competitor for rapid evaluation and application port onto complex parallel architectures. The Parallel and

Real-time Embedded Executives Scheduling Method (PREESM) project, designed by the IETR in the VAADER team, is a rapid prototyping dataflow framework that leverages the modeling of high-level and architecture agnostic dataflow applications. The VAADER team has focused its research work on parallel programming for 18 years. The PREESM project started in 2007, and today it makes the development and automatic parallelization of dataflow applications possible on x86 architectures and multi-core DSPs such as the TI Keystone II.

In this thesis, we study, adapt, and propose new software runtimes and methods based on dataflow programming models to fit this new generation of clustered architectures. We use computer vision, deep learning, digital signal processing, and numerical simulation applications to test the contributions of this thesis.

1.3 Contributions

This section introduces the software contributions introduced in the thesis for a clustered manycore processor. Although this processor is a real machine, manufactured by TSMC in $28nm$, namely the MPPA® processor, all contributions aim at being generalized and applied to similar architectures like the Epiphany-V chip [Olo16] and the PULP accelerator [CRP+16]. Therefore, new techniques for programming the MPPA® processor are introduced in this thesis. Some techniques are adaptations and optimizations of existing methods, enabled for the first time on the MPPA® processor and others are new. The contributions of this thesis are presented at high-level as follow:

- The application of the one-sided communication techniques on a two-sided network-on-chip. Today deployed in production, the new communications engines abstract the Kalray's network-on-chip through high-performance and low-latency one-sided communication primitives, but also, two-sided primitives. The runtime part, the sharing of hardware resources and the ordering of communications are explained. It is built for performance, and it provides a total abstraction of the hardware. An important feature of this contribution is that the provided set of primitives can run asynchronously (running in the background), making this contribution challenging to implement, debug and validate (asynchronism is complicated). Moreover, one-sided communications revolutionized the way of programming the MPPA® processor using explicit communications. As such, it is also the foundation of all other contributions of this thesis.

- An optimized multi-threading runtime to make fine-grained multi-threading possible on the multi-core CPU (cluster) of the MPPA® processor. The runtime enables in the bare-metal Kalray's toolchain the well-known POSIX thread primitives and OpenMP multi-threading. The contribution uses lock-free mechanisms to enable efficient multi-threading. This runtime has production maturity, and, it is used in almost all other contributions of this thesis as the primary operating system running in the multi-core CPU of the MPPA® processor.

- Based on a hierarchical dataflow model, a strategy to generate efficient code for the MPPA® manycore is presented and detailed. This hierarchical dataflow model that was proposed and designed by the IETR's VAADER team, is the key to the contribution. The strategy makes the use of two levels of parallelism possible and efficiently exploited. The two levels of parallelism are the intra multi-core CPU level, and the inter multi-core CPU level. The graph hierarchy of the model enables fine tuning

at early stages of the dataflow application development. The first software synthesis runtime back-end for the MPPA® processor is also presented. Fine speedups are obtained and a drastic reduction time of the dataflow graph compilation is also shown.

- The adaptation of a multi-thread embedded reconfigurable dataflow runtime is provided in the thesis, created by the IETR's VAADER team. The embedded runtime performs just-in-time scheduling of dataflow applications on embedded System-on-Chip (SoC). In this contribution, the original runtime presented in paper [HPD+14], is re-thought to make it run on the MPPA® processor. Most of the work consisted in distributing the runtime on the set of multiple multi-core CPUs implementing limited local memories. The scheduler and the memory allocation techniques have also been redesigned.

- A distributed embedded framework for enabling low-latency execution of OpenVX application graphs is provided. OpenVX is a modern computer vision and Convolutional Neural Network (CNN) inference standard [G+17] to deploy computing pipelines from a host to one or several accelerators. The proposed framework is a clean room implementation of the OpenVX standard specification for the MPPA® processor. The framework runs in standalone mode for the MPPA® processor, making the reconfiguration of OpenVX application graphs possible at runtime. The framework proposes and adapts new techniques to optimize the execution of the OpenVX user graph automatically and at runtime (embedded). Optimizations focus on the reduction of the main memory bandwidth that is most of the time the performance bottleneck. Therefore, the use of automatic kernel fusion and automatic data prefetching over the Kalray's network-on-chip at multiple multi-core CPU levels are used. The framework targets low-latency execution (no batching) to provide a reactive system (for embedded decisions). Results show super-linear speedups at multiple multi-core CPU levels showing that the strategy is effective for clustered manycore architectures like the MPPA® processor.

- Diverse applications using the software runtime contributions of this thesis are provided. These applications are low-level implementations of numerical simulation application based on stencil computation, digital signal processing applications, computer vision or CNN inference applications. In these contributions, parallelization strategies are explained, and how their implementation is adapted to the MPPA® processor. The use-cases are well-known and fundamental in parallel computing. Moreover, some implementations are quite complex since they target the MPPA® processor at low-level. But still, thanks to the previous contributions of this thesis the implementation of these applications are made possible and easier to be both implemented and debugged.

The highlighted contributions are either part of academic researches for the IETR's VAADER team, projects (like the Mordered project) or have direct production values for the Kalray company. The contributions were designed, discussed, implemented and tested under the supervision for the IETR's VAADER team and the Kalray's engineering and research core.

1.4 Outline of this Thesis

This thesis is organized into two main parts. Part I presents the background that introduces the state-of-the-art of our research area and the diverse problems tackled in this thesis. Part II gives and evaluates all contributions proposed in this thesis, that are designed and elaborated for clustered manycore processors.

Chapter 2 defines embedded parallel systems for high-performance computing. The main aspects of the optimization of the performance of an application are given and explained. This chapter also details and presents the MPPA® manycore architecture that is the target of this thesis. Chapter 3 lists and explains standard programming models to program and abstract the deployment of parallel applications onto MPSoCs. This chapter adds and formally explains diverse dataflow programming models. Chapter 4 provides the state-of-the-art of communication technologies for parallel and distributed computer system architectures. Moreover, it provides keys to understand and handle the memory consistency and coherency of parallel architectures with complex memory models.

Chapter 5 explains and describes in details the design of an asynchronous communication library over the Kalray's Network on Chip (NoC) of the MPPA® processor. This contribution is the pedestal of all next chapters of contribution presented in this thesis, and it is designed for high-throughput and low-latency execution. Chapter 6 provides a multi-threading runtime to enable threads on a Symmetric Multi-Processor system (SMP) machine at low-level. This multi-threading runtime is also the pedestal of most further chapters as it allows for efficient fine-grained multi-threading in the Compute Cluster (CC) of the MPPA® processor. Chapter 7 presents a strategy to target and generate efficient code for clustered manycore architectures using a dataflow model as an application input representation. The chapter explains the technique that exploits a hierarchical dataflow model to enable efficient usage of several hierarchical levels of parallelisms of the targeted machine. Chapter 8 introduces an adaptation of an embedded reconfigurable dataflow runtime for the MPPA® processor. The embedded runtime operates on the platform in a standalone way. Chapter 9 introduces the OpenVX standard, designed for efficient computer vision and Convolutional Neural Network (CNN) inference applications with embedded constraints for low-power MPSoC. This chapter proposes an implementation of an OpenVX framework for the MPPA® processor. The framework is built to target the low-latency execution of OpenVX applications. The optimization of the execution latency provides shorter time reaction, that is important for an embedded system running in a car for instance. Chapter 10 presents the implementation and optimization of diverse applications onto the MPPA® processor. Each application is explained and detailed, as well as their parallelization techniques to exploit the processing capabilities of a highly parallel machine. Chapter 11 concludes the work of this thesis and presents future contributions and researches to make the programming of clustered manycore architectures easier and with performance.

Part I

Background

CHAPTER 2

Embedded Parallel Systems

This chapter presents an overview of embedded parallel systems. Several levels of parallelisms at the level of the processing units and the memory systems are highlighted. The Kalray Multi-Purpose Processor Array (MPPA)® processor is presented in details as it is the experimental target of this thesis. A software background is also introduced to illustrate how hardware architectures are programmed for decades and the issues when targeting new embedded parallel systems.

Figure 2.1 – *Software Architecture*

Figure 2.1 shows the classical software architecture that is used in most computer systems. Firstly, the hardware architecture is found which is composed of memories, computing resources, and input-output peripherals. Secondly, the Hardware Abstraction Layer (HAL) [PJ09] and the firmwares are shown, and they provide a hardware dependent set of low-level functions and features to ease the software development. Thirdly, the Operating System (OS) is a software component which manages the hardware resources, the software resources, and it implements simple services used by all the software on running the targeted hardware architecture. Finally, the user application is found, and it runs on hardware and software resources provided by the OS.

At the hardware level, the diverse components of the architecture have to operate together to perform one or several computations. Embedded system designers design their solutions usually to satisfy some application constraints. As the embedded world usually requires low-power systems, powered by batteries, the hardware is inherently simpler than desktop processors which consume more than 100 Watts. Thus, it is harder to get performance on embedded architectures as the hardware will not catch up with the bad software

implementation. Furthermore, as applications are becoming more and more complex, embedded systems require more and more heterogeneous computing capabilities to handle different kinds of processing. Thenceforth, complex embedded systems are confronted with a new parallel computing problem that is: the system heterogeneity.

At the software application level, many programming models exist, due to the diverse hardware architectures, their evolutions, the various programming environments provided by Multiprocessor System-on-Chip (MPSoC) vendors, and new ideas of researchers trying to simplify the programming of MPSoCs for application engineers. Programming these new architectures is a real challenge for all computer science scientists who need to exploit the hardware parallelism at a high-performance to cope with the ever-increasing computing workload of modern applications. A lot of standard and non-standard programming models are currently proposed, showing that it is an unsolved problem. Some of these programming models are listed and explained in Chapter 3.

In this thesis, we provide new runtimes and methods to ease the application development on a new generation of clustered manycores. To do that, we contribute to the firmwares, the OS, and the application layers targeting the Kalray MPPA® processor.

General notions on embedded parallel architectures are presented in Section 2.1. The memory architectures and the memory hierarchy are introduced in Section 2.2. The massively parallel Kalray MPPA® architecture is described in details in Section 2.3. Finally, we explain how the software in Section 2.4 uses these hardware platforms.

2.1 Embedded Parallel Architectures

Embedded parallel architectures are usually low-power and not made for general purpose computing as seen previously. As applications need more and more computing power, parallelism is required since we can no longer increase the computing frequency of Processing Elements (PEs) composing the (embedded) systems. That is why modern computing systems are parallel and heterogeneous.

2.1.1 Multiple Level of Parallelisms

Several types of hardware parallelism exist. The Flynn [Fly72] taxonomy describes and classifies the level of concurrency of computer systems. Only the most important categories are explained in this section, and each type is illustrated in figure 2.2. The idea is to classify computing architectures by their ability to manipulate concurrently (or not) the data and instructions.

The simplest is the Single Instruction, Single Data (SISD) which takes a single input with a single instruction and produces a single output, it is also known as a non-vectorial scalar Central Processing Unit (CPU).

The Single Instruction, Multiple Data (SIMD) instruction operates onto a vector of register(s) and perform the same instruction onto multiple data inputs providing numerous outputs in a single CPU clock cycle [Rus78]. The SIMD parallelism method is part of the data parallel parallelism. Nowadays almost all CPUs, Digital Signal Processors (DSPs), Graphics Processing Units (GPUs) and hardware arithmetic unit in Field-Programmable Gate Arrays (FPGAs) use the SIMD level of parallelism. SIMD is also known as vector processing.

The Single Instruction, Multiple Threads (SIMT) (not part of the Flynn taxonomy as it is more recent), proposed by the Nvidia GPUs' hardware manufacturer for the first time in 2006, introduces the concept of executing the same instruction concurrently onto multiple

threads. It is an execution model where SIMD is combined with multi-threading and implemented in the Nvidia Tesla GPU presented in [LNOM08]. SIMT is very efficient onto regular code but leads to poor performance when control flows diverge for instance with test statements. However, most recent enhancements of GPUs architectures like Volta [Blo17], introduced in 2017, optimizes the execution of flow-divergence by using independent thread scheduling for interleaving such code statements.

The Multiple Instructions, Multiple Data (MIMD) is applied to most CPU-based multi-core architectures. MIMD features several CPU for executing different instructions onto different input and output data concurrently. MIMD is also found in Very Long Instruction Word (VLIW) core at registers and instructions level. VLIW cores can execute multiple SIMD instructions onto multiple different inputs and outputs in a single clock cycle. Today MIMD defines modern multi/many core architectures where several levels of parallelisms can be found like vectorization and multi-thread execution that can be seen in Figure 2.2.

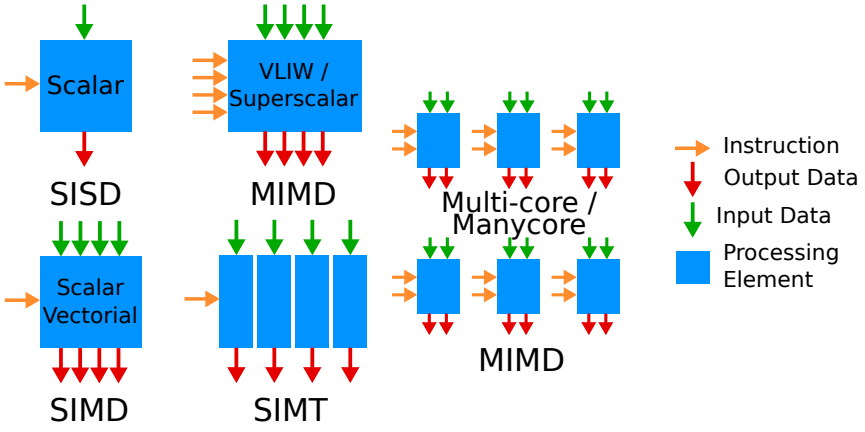


Figure 2.2 – *Examples of Multiple Levels of Parallelism*

Parallel computing is old; in 1975, Cray Research released the *Cray-1* [Rus78] parallel supercomputer. The *Cray-1* made history of supercomputers as it was the first one to implement vector processing. At this time, *Cray-1* was already providing 160 megaFLOPS (Floating Point Operations per Second). Today, MPSoCs implement parallelism at several levels: Instruction-Level Parallelism (ILP), thread-level and cluster-level also known as Non-Uniform Memory Access (NUMA) node. In March 2018, the world fastest supercomputer was the [iW18]. It delivers a measured performance onto LINPACK [DLP03] benchmarks of 93.01 petaFLOPS. This supercomputer exploits SIMD, thread-level, and cluster-level of parallelism.

2.1.2 Heterogeneous Parallel Systems

Embedded systems are becoming more and more involved with a lot of dedicated functions with aggressive performance constraints. Thus, such systems require heterogeneous computing to satisfy all requirements of the embedded computer system. Heterogeneity is defined by having different computing resources in the same systems. For example, a typical heterogeneous system regroups CPU cores, DSP cores, GPU cores, and specialized tightly coupled co-processors in a single chip.

Parallel heterogeneous computer systems are complicated to program, as the same system integrates several computing machines, programming constraints, programming models and with different performance bottlenecks. Moreover, they all need to be orches-

trated concurrently and, if they share resources, those resources have to be managed and protected.

Heterogeneity is found in parallel architectures such as in the Nvidia Tegra X1 [Nvi15] which implements 4 Advanced Reduced Instruction Set Computer (RISC) Machine (ARM) cores and 256 Nvidia CUDA cores, the Texas Instrument Keystone II [Ins17] integrating 4 ARM cores and 8 DSP cores. System heterogeneity can also be found in massively parallel architectures implementing different levels of memory hierarchies (either cached based or local memory based). They are introduced in Section 2.2.1.

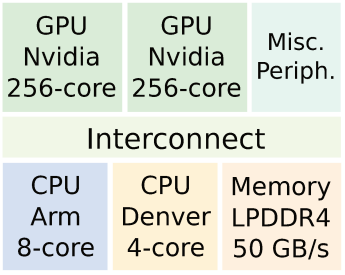


Figure 2.3 – *Nvidia Drive PX 2: A Complex and Highly Parallel Heterogeneous Embedded System for Autonomous Driving*

Another example of a complex heterogeneous embedded system is the Nvidia Drive PX 2 [Nvi17]. It includes two Pascal GPUs of 256 cores each, 8 ARM cores as a multi-core embedded host CPU, and 4 Nvidia Denver CPUs, as shown in Figure 2.3. Such parallel systems are very complicated to program and verify as they offer a high concurrency, both on the hardware and software sides. Such embedded systems are designed to target the autonomous driving (Advanced Driver-Assistance System (ADAS)) which requires substantial embedded processing capabilities.

2.2 Computer Memory Systems

2.2.1 Memory Hierarchy

The memory hierarchy is fundamental in computer systems. If it is not used correctly the performances can be reduced. The common memory resources used in computer systems are the *Registers*, the *Caches*, the *Local Memory* and the *Random Access Memory (RAM)*. In computer systems, the memory latencies are different, and this latency depends on the requested physical memory addresses and the requesting initiators which can be a core, a Direct Memory Access (DMA) or a peripheral.

Definition 2.2.1.1 (Latency)
Latency is defined as the total round trip time to access a resource. The round trip is respectively the sum of, the time to initiate the transaction, the time for the transaction to reach the resource, the time for the resource to process the transaction and the time to return the answer to the initiator. More formally, the latency is the total completion time of an initiated transaction from the start to the end.

The memory latency access time is increased when higher levels of memory are accessed. Different types of memories are described in a commonly known hierarchical order, from low-latency memories (bottom level) to high-latency memories (top level). Since 2005, the memory access latency has not moved, as shown by Berkeley in [Ber18].

We list and describe the typical latency and the purpose of each of these memory levels:

Register is also known as the core *register file*. Registers have a very low latency access time, within a machine clock cycle. Registers are used for internal core computations. They are fed by the system memory, usually from cache or local memories via *Load / Store* instructions.

Caches typically have several levels of hierarchy. Caches offer transparent memory access to the main memory. Typically, on cache hit, the level 1 is 1 ns and the level 2 is 4 ns. However, the Worst-Case Execution Time (WCET) of caches is challenging to be bound, in particular, in multi-core architectures, when the data coherence needs to be maintained among cores with performance [LMW99].

Local Memory or Tightly Coupled Memory (TCM) is an on-chip memory with its own address space. Deterministic response time can be achieved when using local memories. A typical memory access latency is between 1 ns and 4 ns usually depending on the local memory hierarchy. Unlike caches, local memories require explicit software management of data movements. Therefore, the resulting software implementations are more challenging and usually not portable to other architectures.

RAM is a volatile external memory. It has a high latency which is at least 100 ns depending on the memory load. The purpose of lower memory hierarchy levels is to reduce as much as possible core stalls when accessing this memory assuming that the running applications have enough data locality. Examples of RAM technologies are the Double Data Rate (DDR), the Graphical Double Data Rate (GDDR), and the High Bandwidth Memory (HBM) technologies.

Other memories exist, such as physical data storage devices but also remote data storage devices that can be accessed over a network. Remote data storage have a huge latency. A round-trip is in the order of 100 μs.

2.2.2 Memory Architectures

As seen in 2.2.1, the memory hierarchy is what is seen by a core in a memory system, whereas the memory architecture constitutes the entire memory system including how a set of cores are connected to the memories. The memory architecture of a computer system is composed of caches, local, and RAMs that are interconnected with each other on modern systems. The simplest memory architectures are composed of a single memory with a single core. Today most of them are ultra-low power embedded systems implementing a Micro-controller Unit (MCU). In this section, parallel computer memory architectures are introduced. Two principal memory models are presented, the shared memory and distributed memory models. The shared memory model is a unique memory, implementing several memory banks for performance, where cores and peripherals share this memory resource. The distributed memory model implements several memories dispatched over a network. Although distributed memory architectures are more complicated to program efficiently, the motivation for distributed memory is to let massively parallel applications scale onto highly concurrent computer systems.

Uniform Memory Access (UMA) architectures are based on the shared memory model where all cores see a single physical global address space. UMA is used in Symmetric Multi-Processor system (SMP) which is the most used parallel programming model (see Chapter 3 for more details). As several cores can access a unique memory, this memory usually has a full cross-bar to sustain the number of memory transaction requested by cores onto the different memory banks. However, when the number of cores increases, UMA architectures tend to provide poor performance because of the sharing of the memory and memory arbiters, leading to memory access conflicts.

Non-Uniform Memory Access (NUMA) architectures are distributed memory architectures with transparent memory accesses. Transparent memory access means that cores can access the global address space through their cache memory hierarchy. NUMA architectures are easier to program than distributed memory architectures as the data communication between cores is entirely hidden by the diverse levels of data caches which performs the communication automatically. Thus all data communications are done by *Load / Store*, which poorly scales onto massively parallel architectures as the user sees an SMP architecture where processors have very different memory latency access time. It is called the NUMA effect. Moreover, NUMA computer systems usually implement cache coherence, called CC-NUMA architectures, which generates huge coherence traffic when data updates occur (data sharing). As such, using NUMA computer systems efficiently requires to place the accessed memory buffers as close as possible to the computing resources, the cores.

Distributed Memory architectures are composed of an array of memories interconnected with a network. Memories composing this array have computing resources connected to them like cores or custom hardware accelerators. The network can be either a Network on Chip (NoC), an Ethernet network or any custom processor interconnects like the proprietary Intel QuickPath Interconnect (QPI) [Int18]. Such chips are also known as No Remote Memory Access (NORMA) architectures if no hardware or software emulated cache system [KCDZ94] is provided for transparent global memory accesses. Therefore, the programming of these architectures is challenging, and extremely complicated [Ras87] as all data movements must be explicit message-passing initiated by software. However, NORMA architectures make scalability possible when designing highly massively parallel systems. Indeed, computing resources are isolated from each other, making conflicts to access shared resources almost null; thus, allowing scaling.

Heterogeneous Distributed Shared Memory architectures include previously described memory architectures which are the UMA, NUMA and NORMA architectures but exposed at different hierarchical levels in the computer system. The UMA model is exposed at the multi-core CPU level; usually, less than 16 physical cores (nowadays), implementing a shared data cache for transparent memory accesses and/or a single or several (shared) local memories. The NUMA model is exposed at multi-cluster (a cluster is a multi-core CPU) level, but the global address space is unified. Also, most of the time, a cache coherence protocol is provided at multi-cluster level. However, as very large NUMA systems scale poorly because of the *Load / Store* protocol (see Section 4.3.1), the NORMA model is then used with explicit communications. Such hierarchical memory models are more complex to be programmed because both the hardware and the software are heterogeneous. The hardware heterogeneity is due to the exposition of the shared and distributed memory model. The software heterogeneity is due to the use of the SMP model at multi-core CPU

level and of explicit data movement over the network to communicate between multi-core CPUs (clusters).

2.3 MPPA® Manycore Processor

MPPA® is a processor developed by a French company, called Kalray. Kalray is a fabless semiconductor company created in 2008. It is a spin-off from the Centre des Energies Atomiques (CEA). Kalray designs both the hardware and software of the MPPA® processor. On the hardware side, both the front-end and the back-end are delivered: the design, the validation, and the placement routing. On the software side, Kalray provides the compiler, the debugging tools, the operating system ports, the micro-firmwares, the programming environments, the optimized libraries, and the frameworks.

In this section, we focus on the architectural details of the MPPA® manycore processor as they are required to understand the challenges and contributions of this thesis. The overview of the architecture is presented in Section 2.3.1 together with the memory hierarchy. The computing resources of MPPA® are presented in Section 2.3.2 and the communicating interfaces are detailed in Section 2.3.3. A short presentation of the underlying low-level software is given in Section 2.4.5.

2.3.1 Architecture Overview

The Kalray MPPA® (Massively Parallel Processor Array) manycore architecture is designed to achieve high energy efficiency and deterministic response times for compute-intensive embedded applications [SEU+15].

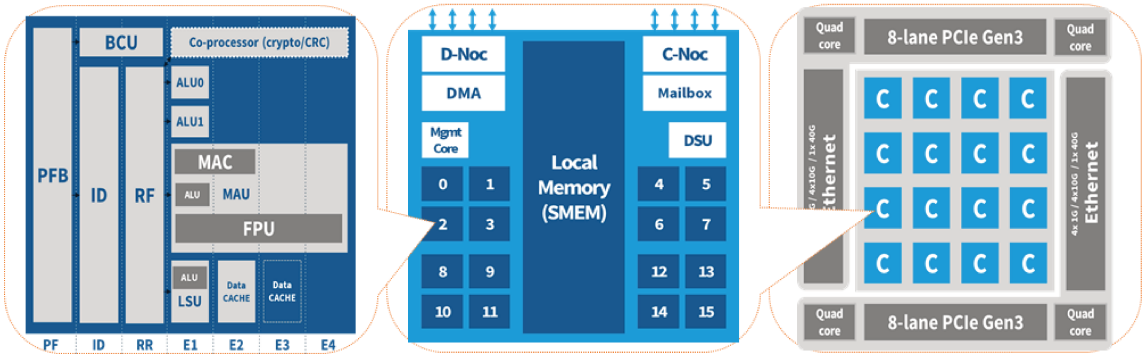


Figure 2.4 – *MPPA® Processor*

The second generation of the MPPA® processor, seen in Figure. 2.4, code-named Bostan, integrates 256 VLIW application cores and 32 VLIW management cores, 288 cores in total, which can operate from 400 MHz to 600 MHz on a single chip and delivers more than 691.2 GigaFLOPS single-precision for typical power consumption of 12W. The 288 cores of the MPPA® processor are grouped in 16 Compute Cluster (CC) of 17 cores and two Input/Output Subsystem (IO) of 8 cores to communicate with the external world through high-speed interfaces via the PCIe Gen3 and Ethernet 10 Gbits/s.

Memory Hierarchy

Besides register files, MPPA® has a three-level memory hierarchy. The first level is the data cache allowing transparent cached accesses to the second level. This second level is

called the local shared memory which is an on-chip high-bandwidth and low-latency local memory. The third level is the main global memory which is a DDR3 technology. The second level of memory of MPPA® can be configured, either to cache the third level of memory (software emulation of L2 cache using the Memory Management Unit (MMU) inspired from [KCDZ94], like in conventional cache-based systems), or by default as a local memory where the buffers are moved explicitly by software configured DMAs. The third level can also be accessed by IO DMA interfaces or through the IO core L1 data cache by *Load/Store*. Finally, on compute clusters, L1 caches are not coherent between cores and DMA interfaces' writes; thus, the memory coherency is managed by software using full memory barrier, partial memory barrier or uncached memory accesses are used.

Memory Map: An Array of Distributed Local Memories

The hardware exposes a heterogeneous memory map of 20 address spaces (2 per IO and 1 per CC). The MPPA® processor implements a distributed memory architecture, with one local memory per cluster. I/O cores access their local SMEM and private DDR via *Load/Store* and by DMA interfaces. Compute clusters can also access their local SMEM but not the DDR via *Load/Store* and by their DMA interface. The DMA interface must be used to build up NoC packets and send them to the NoC to communicate between the 20 address spaces available.

2.3.2 Computing Resources

k1-Bostan VLIW Core

Each MPPA® core implements a 32-bit VLIW architecture which issues up to 5 instructions per cycle, for different execution units: branch & control unit (BCU), ALU0, ALU1, load-store unit (LSU), multiply-accumulate unit (MAU) combined with a Floating Point Unit (FPU). Each ALU is capable of 32-bit scalar or 16-bit SIMD operations, and the two can be coupled for 64-bit operations. The MAU performs 32-bit multiplications with a 64-bit accumulator and supports 16-bit SIMD operations. Finally, the FPU supports one double-precision fused multiply-add (FMA) operation per cycle or two single-precision operations per cycle. SIMD instructions are supported by the FPU to accelerate classical floating-point computations (adds, multiplications, and complex calculations).

IO Subsystems

Each IO integrates two quad-cores. Each quad-core implements 4 cores of VLIW architecture as previously explained. Then IO subsystems are connected to a 4 GB of external DDR3 memory and on-chip Shared Memory (SMEM) memory of 4 MB. Regarding memory accesses of cores, cached and uncached accesses can be performed for both Load and Store operations (64-bit/cycle) in the SMEM and DDR3. For the shared memory, cached and uncached atomics are available such as Load-and-Clear, Fetch-and-Add, and Compare-and-Swap (CAS). Atomic cached operations provide execution efficiency when dealing with critical algorithmic parallel paths that need mutual exclusion or atomic updates of variables. Each IO embeds 8 high-speed IO interfaces usually, called DMA, to communicate through PCI Express First-In-First-Out queues (FIFOs), Ethernet, DDR3 and SMEM. The PCI Express implements 16 lines of 8 Giga Transfer per Second per line; therefore, a single line provides roughly 1 GB/s. As such, the maximum full-duplex PCI Express theoretical bandwidth is up to 16 GB/s with the PCI Express DMAs. Finally, the software

is in charge of maintaining the memory coherence between DMA reads/writes (for both PCI Express and the NoC) and the cores.

Compute Clusters (CCs)

Each CC embeds 17 cores, 16 PEs and a Resource Manager (RM). CCs integrate a multi-banked private local SMEM of 2 MB. Memory accesses of cores are supported only in this SMEM, and only uncached atomic instructions are available. The same atomic instructions are available in the IO subsystem as presented before. Each CC has one DMA interface for communicating with external nodes. Here, the software is also in charge of maintaining the memory coherence between DMA reads/writes and the cores.

2.3.3 Communications

Network-on-Chip

A full-duplex 32-bit wide NoC interconnects the 18 multi-core CCs (CPUs) of the MPPA® processor. The NoC implements wormhole switching, with source routing, and supports guaranteed services through the configuration of flow injection parameters at the NoC interface: the maximum rate σ; the maximum burstiness ρ; the minimum and the maximum packet sizes (the size unit is the flit). A flit is 32-bit (4 bytes per cycles), meaning a bandwidth of 2 GB/s per link direction when operating at 500 MHz. The NoC is a direct network with a 2D torus topology. This network does not support Load/Store but only data NoC stream and low-latency control NoC messages. Thus the software is in charge of converting virtual memory addresses to the data stream (data NoC), and of converting this stream back to the virtual address in the remote memory to initiate any communications between any multi-core CPUs.

Control NoC Interface

The control NoC is made to communicate at very low-latency with 64-bit messages. It does not have access to the memories (on-chip or off-chip memories); the messages are mapped in the DMA interface registers. Each DMA interface implements 128 64-bit control NoC receive mailboxes (Rx) and 4 transmission resources (Tx). These mailboxes can be used for barriers and simple 64-bit messages with a notification on a list of processors (up to 17 cores in CC). The barrier mode is mainly used for generic inter-core low-latency synchronization and notification. For instance, forcing a remote core or a poll of remote cores out of idle state in a single clock-cycle for the initiating core. A store in the peripheral space is a posted operation. The 4 Tx resources must be shared between the cores of the multi-core CPU. A NoC route and a remote control NoC Rx mailbox identification number (called a tag in the range $[0, 127]$) must be configured to send a 64-bit message through each control NoC Tx resource.

Direct Memory Access NoC Interface

The data NoC feature is made for high throughput. Therefore, it is a very asynchronous hardware block that requires to be handled **asynchronously** by the software. Indeed all outstanding incoming and outgoing transactions must be managed by the software asynchronously for performance. Each data NoC DMA interface is composed of three elements:

- Eight micro-cores, running concurrently, are available for each DMA interface. A micro-core is a micro-programmable DMA core that needs to be programmed and configured. It has a simple set of instructions such as *reads*, *local* and *remote notifications* for local and remote completions and added support for the arithmetic of internal read pointers and counters. It can execute up to 4 nested loops to describe custom memory access patterns with high throughput. This throughput is limited by the technology of the memory on which the micro-core is reading, the NoC link size (4 bytes/cycle) and the memory access patterns.

- The data NoC implements 256 Rx Tags (range $[0, 255]$) per DMA interfaces to write incoming data NoC packets in the local memory of compute clusters or in the DDR memory of IOs. This Rx Tag has a write window described by a base address, a size and a write pointer that need to be configured and managed at runtime. The completion of the incoming data transfer is given by an End-of-Transfer (Eot). This Eot command increments a 2^{16}-bit notification counter corresponding to the used Rx Tag in the DMA interface of the MPPA® network.

- Each DMA interface implements 8 packet-shapers. A packet-shaper (DMA Tx) is a hardware unit that is building data NoC packets using data coming from a PE or a micro-core. Then, the packet-shaper sends these NoC packets in the MPPA® NoC using the configured NoC route. Indeed all NoC routes and injection parameters Quality-of-Service (QoS) need to be set by software.

2.4 From Parallel Architectures to Software

2.4.1 Operating Systems

An operating system is a low-level software that controls the hardware resources, makes tasks scheduling possible and allows multiple processes to share diverse resources such as the peripherals and the memory. The operating system is usually ported or developed at bare-metal level. It can also be ported at the hypervisor level also known as Virtual Machine Monitor (VMM) when targeting a virtual machine but it is out of the scope of this thesis.

Bare-metal System

The so-called "bare-metal" system is directly based on the hardware without any runtime or virtual support (ie the VMM). The programmer is in charge of everything such as powering up the System-on-Chip (SoC), enabling instruction and data caches, enabling the MMU, handling system interrupts and hardware trap exceptions, configuring FPU rounding mode, setting up stacks of threads and many more architectural details related to the underlying SoCs. Therefore, such mode cannot be used of portable software as most written software is dependent on the hardware. Bare-metal programming is used for porting OS, hypervisors or low-level runtime firmwares onto SoCs. Such a level of programming can only be used by SoC experts, usually hardware providers.

Real-Time Operating System

A real-time OS is designed to meet real-time constraints: computation deadlines must be met. Embedded systems usually use a real-time OS. Programmers have strong control over

timings regarding the scheduling of tasks and external events mostly managed through interrupt handlers. Such OS implements ready-to-use synchronization and message-passing primitives between tasks. However, efficient multi-core implementations of such an OS is not a simple task. A locking mechanism for mutual exclusion must usually protect the shared resources. For instance, the scheduling task is inherently sequential in [MNW14] ("Scheduler's lock"). Thus with fine-grained parallelisms, speedups can be poor because of system runtime overhead. Indeed, most multi-threading implementations use locks for managing threads scheduling and liveness, making the serialized software section a performance bottleneck, as in [MNW14], *GLIC* [SLwRMSD18] and [HPD+14] x86.

Linux Operating System

Linux is one of the most used operating systems of the open source community [TRBD01]. It currently targets most mainstream computer architectures. The name *Linux* is used to denote the entire system: the kernel and all user applications or system software interfaces running on top of the kernel [Tor97]. A powerful feature of the Linux kernel is that it cannot (should never) be corrupted by user software. The isolation between the user-space and kernel-space is performed with hardware and software (IO-)MMU mechanism. MMU entries are only filled by trustworthy kernel software on page fault exception. The communication between user applications or system interfaces within the Linux kernel is done using explicit Input/Output Control (IOCTL) functions, also known as system calls. System calls are used to invoke kernel services or kernel machine specific drivers, for instance to deal with Peripheral Component Interconnect Express (PCIE), Universal Serial Bus (USB) or Ethernet interfaces. The Linux OS has been designed to operate onto parallel machines like SMP and NUMA architectures. Linux manages the memory consistency of the data (cached or un-cached), the instruction caches for relocated code, and page table mappings for the virtual memory. Such features make Linux a great candidate for software system portability onto complex parallel machines. Also, Linux provides a preemptive scheduler, with task scheduling priorities or thread pinning to cores using thread affinity special attributes that are provided by the execution runtime.

2.4.2 Classical Software Memory Layout

The memory contains all the executed instructions and user data of a program running on one or several cores. Memory accesses are vital to the computer system performance, as they are on the critical path of the computer system. Although the number of accesses to instructions is higher than reads or writes in the memory, still application data accesses are crucial for performance [WM95]. Figure 2.5 shows an embedded multi-core computer program operating onto a memory. Cores can access the main memory through their memory hierarchy that is presented in Section 2.2.1.

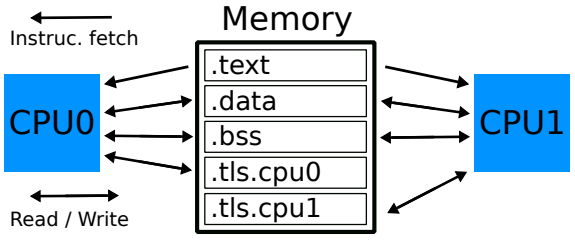


Figure 2.5 – *Typical CPU Cores Linked to a Memory with Memory Access examples*

In the standard Executable and Linkable Format (ELF), the data segment nomenclature shows the *.text* section that contains the executable machine code (instructions) of the program which is shared and read-only in statically linked programs. The *.data* and *.bss* sections of the program can be accessed by the read and write operations. These memory sections are initialized for the *.data* and non-initialized for the *.bss*. The *.tls* section stands for the Thread Local Storage (TLS) [Dre03] which can only be accessed by a core owning it by read and write operations. In standard compilers, like GNU Compiler Collection (GCC), the TLS is used using the $\_\_thread$ attribute to the object declaration. An instance of these objects is then replicated in each thread's TLS memory. Also, each core has its own stack which contains variables of functions, and the heap is then found if a dynamic allocator is available [WJNB95]. The stack and heap are at the end of the *.bss* section or constitute an entire section by themselves in the memory. The stack and heap are accessed by core using reads and writes operations.

All data segments are placed in the memory, which is concurrently accessed by all running cores. The user program itself defines memory accesses. Accesses are either static, dynamic or both depending on code sections. Static accesses are easily predictable, but dynamic accesses will rely on the input data or complex addressing within computation loops. Moreover, data reads and writes to the memory are usually more random than instruction fetches that are easier to predict.

For years, computer scientists have been investigating methods to provide programs with efficient memory accesses [SHW11]. However, this is a complex task as it strongly depends on the application and its implementation. The computer memory systems are sometimes misused, even if the hardware and the low-level software runtime implement advanced mechanisms to compensate bad software application implementations (for instance the irregular memory accesses). This hardware, software, or hybrid mechanisms are explained below in this chapter.

2.4.3 Software Management of the Virtual Memory

Most processors operate in memory using virtual addresses, that are decoded by a MMU to get the physical addresses, to send the memory request to the proper physical memory or memory banks. Rich (most standard) operating systems use hardware and/or software MMU as the Linux OS presented in Section 2.4.1. The MMU is a hash table which maps virtual addresses to physical addresses. Figure 2.6 shows where IOMMU and MMU are placed in a heterogeneous computer system and how they do interact with other units. For performance, the MMU implements a Translation Lookaside Buffer (TLB), which is a memory cache that can be hierarchical. The TLB associates the requested virtual address to a physical address. If the requested virtual address is not mapped in the TLB, then a hardware page fault exception is generated to the CPU. On a hardware page fault, which is typically a TLB miss, the operating system takes over. It can either write a new entry in the TLB from the software managed MMU, or stop the execution of the program if it is a user-space segmentation fault. The segmentation fault occurs when the user-space requested an address that is invalid memory access due to buggy software such as the dereferencing of a $NULL$ or corrupted pointer, or a stack overflow.

2.4.4 Software Concurrency

As most of the software written in this thesis runs on a manycore architecture, it is essential to define the software concurrency.

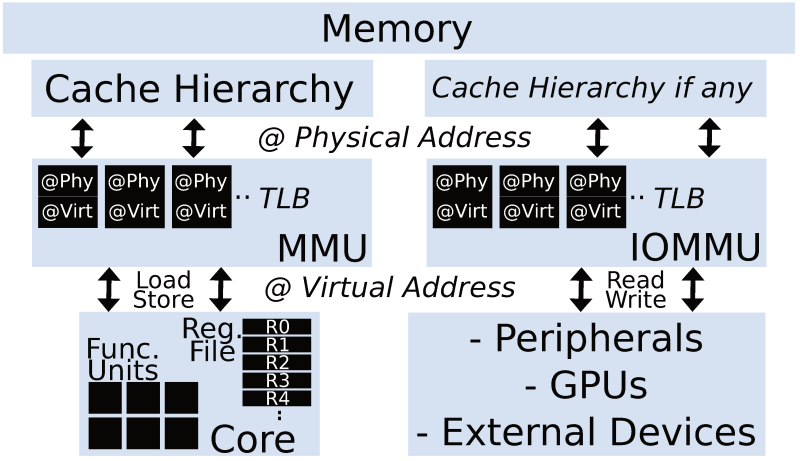


Figure 2.6 – *(IO)MMU Role in a Heterogeneous Computer System*

Concurrency is related to computer systems that can execute a set of tasks or instructions simultaneously onto different hardware or software resources. As in paper [Sut05], concurrency defines software composed by a set of tasks that can be executed in parallel or out-of-order without affecting the outcome. It supposes that the result is not affected by race conditions or timing anomalies due to concurrent executions.

Concurrent software is difficult to design as it is error-prone and the global number of reachable states is the product of all states of individual tasks composing the concurrent software. Thus, it has a complexity of $O(\prod_{i=1}^{N} Si)$, with N the number of concurrent tasks executed in parallel and Si the number of states of each task composing the software. The complexity of parallel software is exponential with the number of components composing it.

Concurrency can be found at several levels as explained in Section 2.1.1. Concurrent software is rarely fully parallel. There are parts of concurrent systems that require control or initialization time which are usually sequential tasks. Moreover, parallel software faces new problems like consistency points between sets of tasks and the management of shared resources. The consistency between parallel tasks is probably the biggest problem of concurrent executions.

Well-known forms of parallelism in parallel programming are the data and task parallelisms. The data parallelism defines the concurrency between multiple instances of a single operation onto multiple inputs and outputs. Usually, data parallelism has a fine degree of parallelism. The task parallelism defines concurrency between multiple instances of different operators. A task is generally related to coarse-grained execution where each task performs various operations.

2.4.5 Available Software for the MPPA®

This section explains the software level layers where most runtime tools presented in this thesis have been implemented. Developing at this level onto MPPA® is called: bare-hypervised development. Figure 2.1 shows the architecture of the low-level software when an OS and user-space application are built. The classical software architecture is shown in Figure 2.1 is not only related to the Kalray MPPA® processor, but it is also true for other processors.

The Kalray Exokernel

The Kalray exokernel has been designed and implemented by Pierre Guironnet de Massas, Ph. D. [dM09], is the lowest software layer of the Kalray MPPA® processor. The exokernel is situated at the same level of the hardware layer in the Figure 2.1. The goal of the Kalray exokernel, also called hypervisor, is to prototype in software virtual hardware features such as smarter DMA, caches [KCDZ94], spatial partitioning [MRPC10], and MMU virtualization. When the implemented software features in the Kalray exokernel have maturated and have become a bottleneck for the targeted applications, they can be implemented in hardware in next chip generations. Virtualization is an essential feature for partitioning, system isolation and debugging as parallel systems are becoming extremely complex; therefore, the confinement of bugs or system failures is mandatory. Developing right above the Kalray exokernel interface is also known as bare-hypervised level of development. We use below the bare-hypervised level to describe the lowest and standard level of system development on the Kalray MPPA® processor.

HAL Level

The HAL [PJ09] provides an abstraction of the hardware features and it can be hierarchical. Usually, the functions of a HAL are hardware dependent. When this is huge architectural changes between two versions of a chip, the HAL might genuinely change. On MPPA®, the HAL simplifies the programming of the DMA NoC interface, for instance for configuring NoC routes, NoC bandwidth limiters, DMA micro-engines or receiving channels. Also the HAL provides functions to use the k1-VLIW core more easily, such as getting the CPU or CC identifier, managing events and system registers.

OS Level

The Operating System (OS) is usually running over the Exokernel. The OS provides the implementation of system calls, tasks, resource management, Processing Elements (PEs) control for the software threads, synchronizations, interrupt handling atomicity when accessing the hardware from the user level. The OS cannot corrupt the Exokernel. For performance, the OS is able to access the MPPA® hardware directly, but the Exokernel has already set the access rights.

Applications & User Libraries Level

The applications run on the OS and can invoke OS services through system calls. A system call is redirected to the Exokernel which then redirects it to the OS. If the OS does not implement MMU protection mechanisms, and if the application is failing due to a software bug, the application may corrupt the OS (not true on Linux for instance) but not the Exokernel. Indeed, the Exokernel has always protected thanks to the MMU. For performance, the application is able to access the MPPA® hardware directly but the Exokernel and the OS have already set the access rights.

Software Emulated Distributed Shared Memory (DSM)

As the Compute Clusters (CCs) of the MPPA® processor do not have hardware support for accessing the external off-chip memory, for programmability and the support of the OpenCL-C memory model, a transparent memory access mechanism through *Load-Store* was required. Also designed and implemented by Massas [dM09], the software emulated

L2 data cache is called the Distributed Shared Memory (DSM). The Distributed Shared Memory (DSM) system, inspired by [KCDZ94], uses the MMU trap (miss) mechanism to perform data page refill in the off-chip memory.

A part of the local memory of the CC serves as a 'cache' to store the MMU pages, transferred explicitly by the software configured DMA NoC interface. On TLB miss, if another PE already holds the page within the CC, the TLB entry is immediately written. If not the PE sends a request to the IO. Then the IO performs the refill. When the data arrives at the PE, the PE returns from the MMU page fault handler and goes on with the execution (called Return From Exception). Such software mechanisms are extremely complicated to develop, debug, and validate. They are highly concurrent, with asynchronism, and a lot of transitional states need to be handled.

2.5 Conclusion

This chapter introduces general notions about parallel computing and presents the Kalray MPPA® processor. We define parallel embedded systems in the context of high-performance computing. Multiple levels of parallelisms are exposed, as the targeted processor in this thesis is the Kalray MPPA® architecture.

The MPPA® architecture implements multiple levels of parallelisms in the same SoC. They are the SIMD (vector processing), ILP (VLIW), thread level (multiple PEs in a CC), and the process level (multiple CCs in the same SoC). They are all used in the contribution part of this thesis. We provide low-level details about the memory hierarchy, the CC, the IO, the NoC and the DMA NoC interface. The NoC and the DMA NoC interface explanations are essential for understanding the contribution in Chapter 5.

We also highlight issues encountered in parallel computing that are the main memory bandwidth bottleneck (the memory wall), and the heterogeneity in the memory access latency, memory types, and the available computing resources. Finally, issues in the design and development at the system level are presented, like the management of the virtual memory address space (Linux Kernel, Exokernel protection), the memory map of the standard ELF, and diverse OSs running in SMP mode in the compute nodes of MPPA® (CC or IO).

One of the hidden biggest issues seldom mentioned is the debuggability and the observability of the parallel system implemented on such a heterogeneous parallel machine, namely the MPPA®. Being able to debug, observe, and understand why the system is failing requires in-depth knowledge of the low-level architecture when developing at the system level. A lot of system level debug has been done while building the contributions of both Chapter 5 and 6, without which the other contributions of this thesis would not have been possible. Indeed, the complexity of the contributions presented in this thesis is limited by my capacity to debug them. This is very difficult in a massively parallel environment that needs to implement asynchronous transactions for performance, especially at the system level.

CHAPTER 3

Parallel Programming Models

This chapter introduces parallel programming models usually used for programming Multiprocessor Systems-on-Chips (MPSoCs). One of the main goals of a programming model is to hide the complexity of the targeting hardware. Several programming models exist, and it is due to the diversity of applications and hardware architectures. But still, today two kinds of programming models are identified. The first one is the multi-threading programming model using a Symmetric Multi-Processor system (SMP) with a flat memory hierarchy model (single address space). The second one is the acceleration programming model where a host offloads some computations onto one or several accelerators. Many programming models have been proposed for decades, and still, engineers and researchers are looking for new ways to exploit and describe applications for MPSoCs efficiently. Once again, this is due to the ever-increasing hardware complexity, system heterogeneity, and new applications that are also becoming more and more complex. This chapter aims to focus on existing standard or non-standard parallel programming models to target parallel machines.

The Section 3.1.1 presents the task programming model using an SMP architecture. The most commonly used models are explained in this section such as OpenMP multi-threading and the Pthread programming Application Programming Interface (API). Acceleration programming models are presented in Section 3.2. On the targeted clustered architecture, the most used programming model is the acceleration because applications require most of the time centralized control to manage input and output data. In this section, we explain the purpose of such an execution model, and we describe the OpenCL programming model currently available on the targeted clustered manycore in this thesis. Section 3.3 presents dataflow programming models. We show the advantages and fundamental properties of such models. For instance, the hierarchical dataflow models are introduced, that are important to target hierarchical machines like the Multi-Purpose Processor Array (MPPA)® processor. We also give state-of-the-art about mapping/scheduling and memory allocation of dataflow applications onto parallel processors. Fundamentals about scheduling and memory allocation are presented in Section 3.4, not only in the context of dataflow applications but also in case of general purpose computing. In Section 3.5, the rapid prototyping is presented through diverse tools and models. In this thesis, the focus is put on the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) framework and the Synchronous Parameterized Interfaced Dataflow Embedded Runtime

(SPIDER) embedded runtime. But other competitors targeting similar problems are also highlighted in this section.

3.1 Task Programming Models

3.1.1 Processes & Threads

A task programming model runs onto an SMP architecture, or an Operating System (OS) exposing an SMP architecture. The multitasking model uses a shared memory model (section 2.2.2) where all tasks run onto cores and see a shared memory address space as shown in Figure 3.1. It must be noted that write operations from one core in the shared memory are seen by other cores depending on the multitasking model and task memory access isolation. However, this common memory address space can either be an Uniform Memory Access (UMA) or a Non-Uniform Memory Access (NUMA) system with a Distributed Shared Memory (DSM) [KCDZ94]. In multitasking, tasks are either processes or threads.

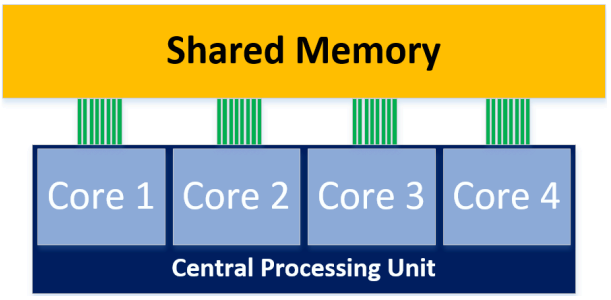


Figure 3.1 – *Symmetric Multi-Processing*

Processes are isolated from each other. They must use the OS kernel to communicate with each other. In this way, processes are heavy to manage for an OS as hardware and software Memory Management Unit (MMU) management are required to provide isolation. But different processes operate independently from an OS point of view even though the hardware, which has limited resources, is still shared.

Threads operate within a process. A thread is a sort of *Lightweight Process*. The scheduling of threads within a process does not necessarily require the intervention of the OS; thus, threads are more efficient for multitasking that involves a lot of task synchronizations and communications. As such, the memory accesses of threads are not isolated from each other within a process, meaning that concurrent executions can sometimes be more complex to develop and more error prone than processes.

Multitasking is known to be efficient for UMA architectures. On UMA architectures, inter-task communications are efficiently executed using *Load/Store* instructions (see Section 4.3.1) natively supported in hardware. Nevertheless, multitasking is less adapted to large NUMA distributed memory systems [KCDZ94]. In this context, inter-task communications based on implicit (i.e. hardware) *Load/Store* instructions and coherency messages traffic on the interconnects are not efficient. The programmer can improve the efficiency at the cost of profound code modifications replacing implicit *Load/Store* by explicit (software calls) *Put/Get* operations (see Section 4.3.2). As such multitask for the parallel system is usually combined with Message Passing Interface (MPI) [HDB+12] or SHMEM [CCP+10]

communications described in chapter 4. The main consequence is then that the optimized software is hardware dependent limiting code reuse and maintainability.

3.1.2 POSIX Threads

The POSIX Threads norm, specified by the IEEE Computer Society was released in the 90s. POSIX defines a portable API to be implemented by the OS or developed over the OS like the [SLwRMSD18]. The Pthread programming model is widely used in SMP programming and implements the threading model as explained in Section 3.1.1. The standard Pthread API is available in/over most multi-core OS like Linux and proposes a large panel of low-level primitives, usable in C programming, to manage threads such as:

- `pthread_t` creation and joining of threads to deal with their life.
- `pthread_|mutex|spinlock|_t` mutex and spinlock for the protection shared resources from concurrent accesses like memory or peripherals.
- `sem_t` semaphores for synchronizations and token consumption.
- `pthread_barrier_t` barriers for collective synchronizations.
- `pthread_cond_t` conditional variables for the implementation of more complex conditions of synchronizations.

Therefore, Pthreads share the same memory space which is accessible by *Load/Store*, and also implement synchronization operations that are required for multi-core programming. The code sample in Figure 3.2 shows how to run multiple threads using the Pthread API. The entry point is the *main* program which performs *pthread_ create* by giving a function *task* with its function argument *arg*. The new threads are scheduled by the OS, and the thread *task* prints the OS thread identifier and the argument value in the range *[0, NB_ THREAD-1]*. In the end, the master thread running the *main* program performs the *pthread_ join* which returns when the corresponding thread *thread[i]* exited. The joining of a thread has two implicit functions, it provides synchronization and memory consistency (Section 4.4.1) management between the exit thread and the thread joining the exit thread.

The Pthread API provides very decent control of the thread resources as it is a low-level multi-threading API. However, Pthread might sometimes be painful to use, especially because of the passing of arguments to threads where data structures need to be written by hand. Other models like OpenMP entirely hide such problems, making them easier to use.

3.1.3 OpenMP Multi-threading

OpenMP multi-threading, is also known as OpenMP 3.0 [CJVDP08], is a portable multi-threading API for SMP architectures implementing a shared memory model. Released for the first time in 1997, OpenMP can be used in Fortan, C/C++ by adding compiler directives that are caught only when OpenMP is enabled at compile time (*-fopenmp*). Thus, it allows code portability when OpenMP is not supported, that is, important for production software which is expensive to modify.

OpenMP 3.0 can be used for both task parallelism (*omp task*), data parallelism (*omp for*). OpenMP 3.0 multi-threading implementations often use the POSIX runtime backend, for instance, the one used in GNU Compiler Collection (GCC). Several and most commonly used OpenMP compiler directives are available and explained as follows.

```
1. #define NB_THREAD (8)
2.
3. static pthread_t thread[NB_THREAD-1];
4.
6. void* task(void *arg) {
7.       printf("Hello thread %ld\n", syscall(SYS_gettid));
8.       return NULL;
9. }
10.
11.int main(int argc, char *argv[]) {
12.      for(int i=0;i<NB_THREAD-1;i++) {
13.               if(pthread_create(&thread[i], NULL, task, NULL) != 0) {
14.                        assert(0 && "Failed create thread\n");
15.               }
16.      }
17.      task(NULL);
18.      for(int i=0;i<NB_THREAD-1;i++) {
19.               if(pthread_join(thread[i], NULL) != 0) {
20.                        assert(0 && "Failed join thread\n");
21.               }
22.      }
23.      return 0;
24.}
```

Figure 3.2 – *Example of Pthread Multi-threading Programming.*

- Management of data: *Shared* data are concurrently accessed and *private* data are replicated within each threads usually placed in the *.tls* section (see Section 2.4.2).

- Synchronizations: provides a fine-grained controlling of the thread resources to deal with data dependency and access shared resources. The following constructs are available:

  Barrier makes it possible to synchronize threads in a parallel region.

  Critical sections are used to scope a code section that should be executed atomically or serialized. Such a code section ends up being protected by a lock. If the section is named, it has its own locking mechanism; otherwise, an unnamed global lock is taken.

  Atomics allow the compiler to generate hardware atomic instructions onto memory access of variables if available and supported in the compiler port. It provides better performance than *critical* sections.

- Scheduling of parallel regions: A *static* schedule will spread the iterations of the work statically onto threads at the beginning of the parallel region. *Dynamic* will decide at each loop iterations on which thread the work will be executed, and, a chunk size can be specified allowing the computation of several iterations at once.

- Reductions: *Reductions* operate on simple operators such as $+, -, *, /$, *min, max*. The *reduction* of a parallel region is applied at the end of a parallel region by combining results of the contributions of all threads. The *reduction* can often be implemented using the *atomic* or *critical* clauses, but it will give lower performances.

A code snippet in Figure 3.3 shows simple OpenMP parallel clauses. In the runtime back-end of GCC (libgomp), the first *omp parallel* directive starts the physical threads

of the parallel region (See Line 10 in Figure 3.3). These threads are neither joined nor canceled, but they stay alive until the process ends. The threads operate within a process. Indeed, the OpenMP runtime creates threads only once to avoid calling the OS each time a parallel region is encountered, as system calls have a huge overhead. The second OpenMP directive (See Line 14 in Figure 3.3) executes the **for** loop in parallel onto the specified number of thread `NB_THREAD`. The schedule is set to dynamic and uses a chunk size of `CHUNK` (See Line 15 in Figure 3.3). In this use case, a static schedule provides almost the same performance as a dynamic schedule when the dynamic chunk is set to *CHUNK* (benched onto a multi-core UMA x86 architecture). Also, the buffers `a, b and c` are `shared` and the `private d` variable, placed on the stack of the master thread, is being replicated in each thread.

```
1. #define NB_THREAD (16)
2. #define CHUNK (8*1024*1024)
3. #define SIZE (NB_THREAD*CHUNK)
4.
5. static float a[SIZE], b[SIZE], c[SIZE];
6.
7. int main(int argc, char *argv[]) {
8.       float d = 0;
9.       /* Starts the internal threads */
10.      #pragma omp parallel num_threads(NB_THREAD)
11.      {
12.              printf("Tid[%ld]\n", syscall(SYS_gettid));
13.              /* Parallel For Loop */
14.              #pragma omp parallel for shared(a, b, c) private(d)
15.                                       schedule(dynamic, CHUNK)
16.              for(int i=0 ; i<SIZE; i++) {
17.                      c[i] = a[i] * b[i] + d;
18.                      d += 1.0f;
19.              }
20.      }
21.      return 0;
22.}
```

Figure 3.3 – *Example of OpenMP 3.0 Multi-threading Programming.*

OpenMP 3.0 exposes a relaxed memory consistency on a shared memory model which is used to provide more efficient memory accesses. For instance, shared variables can exploit uncached (also known as streaming) *Loads* and *Stores* or atomic instructions. As such, each thread has a temporary view of the shared memory which means that it can exploit the cache or a local memory to avoid going to the main memory for every variable reference.

Another feature since OpenMP 4.0 that can be used in OpenMP parallel regions is the vectorization (*omp simd*) that applies the `pragma ivdep` compilation directive. Such directive makes the compiler apply in-core parallelism using vectored instruction.

Kalray Original OS Kalray provides a multi-threading OS called NodeOS. This OS was developed and available since Andey, the first MPPA® generation and presented in [dDdML+13]. NodeOS provides both a subset of Pthread functions and OpenMP multi-threading support. However, the OS has limitations such as the difficulty to get decent performance on fine-grained multi-threading, the impossibility to interleave usage of OpenMP and Pthread multi-threading, the lack of file system support and only up to 16 threads

(one-per Processing Elements (PEs)) is possible. In this thesis, we propose a new implementation of high efficient multi-threading runtime explained in Chapter 6.

3.2 Acceleration Programming Models

3.2.1 Execution Model

An acceleration programming model aims to offload a heavy computational workload onto one or several external computing resources. The acceleration programming model usually implements the master/slaves model where a host (multi-core) Central Processing Unit (CPU) plays the role of a master that deploys computations onto external or remote computing resources playing the slave role.

3.2.2 OpenCL

OpenCL [G+11] is an open source *Computing Language* designed for heterogeneous programmable parallel platforms. The OpenCL standard is cross-platform and was developed initially for CPU and Graphics Processing Unit (GPU) based architectures. The standard was created back at the end of the 2000s by the Khronos Group. OpenCL is well known for its programming flexibility as it gives a strong control of the targeted hardware architectures from a host application and it is said to be close-to-the-metal programming.

As OpenCL is an acceleration API, it involves a host processor and one or several accelerators. In OpenCL, accelerators are called *Compute Devices* which are composed of *Compute Units*. *Compute Units* are composed of *Processing Elements* which performs the computation.

Classical Flow for an OpenCL Application

OpenCL programming can be tedious as a lot of things are in charge of the developer, for instance, an OpenCL application will be described with the following typical sequence:

- Compute Device initialization.
- Compilation of the kernel.
- Creation of input and output buffers.
- Send or map input buffers to the accelerator memory space (Compute Device memory space).
- Set the kernel arguments one by one.
- Send the command of execution to the compute device to execute the kernel.
- Read or map output buffers to retrieve results of the kernel to the host.

OpenCL Memory Model

The OpenCL memory model uses shared memory with multiple levels of memory hierarchy. When writing OpenCL kernels, attributes can be set to the pointer of data. The `__global` attribute specifies a shared data pointer between multiple *Compute Units*. In OpenCL, an issue with global memory accesses (buffers implementing the `__global` attribute) is

the sharing of cached data in multiple *Compute Units*. Known as the false sharing, it implies a performance degradation due to additional data traffic to maintain the coherence. The `__local` attribute defines memory buffers accessible in the *Compute Unit* only and shared across *Processing Elements* in this *Compute Unit*. The `__private` attribute specifies a memory accessible only by the *Processing Element*. As such, with these 3 levels of memory hierarchy directly exposed in the OpenCL language, it makes it possible to tune memory accesses efficiently for the targeted platform. However, as the low-level information is exposed in OpenCL applications, an OpenCL application description will need to be modified when targeting different MPSoCs for performance optimization. Indeed, as the Systems-on-Chips (SoCs), the memory hierarchies and their geometries are different, OpenCL implementations require adaptations for performance. Such adaptations make OpenCL known to be difficult to use, as it requires knowing architecture specificities to get competitive performances.

OpenCL Data-Parallel Support for the Kalray MPPA®

The OpenCL acceleration programming model is possible on MPPA® thanks to the DSM system that provides memory accesses (through *Load/Store*) for the Processing Elements (PEs) of the *Compute Units* to the main global memory of the accelerator, namely the `__global` memory. Moreover, the DSM provides false sharing support when multiple Compute Clusters (CCs) (*Compute Units*) modify the same page. The mechanism is called the *Reconciliation*. However, the reconciliation system has a huge impact on the performance as the software has to perform the byte-to-byte *Read-Modify-Write*. Figure 3.4 shows how the data-parallel model of the Kalray's OpenCL is mapped onto the MPPA® architecture. The memory model is also observed, and it is noticed that the local memory of the Compute Clusters (CCs) is used as a local memory (`__local`, `__private`) and the rest as a cache of pages for letting the Distributed Shared Memory (DSM) operating to access the global memory in the Double Data Rate (DDR) (`__global`).

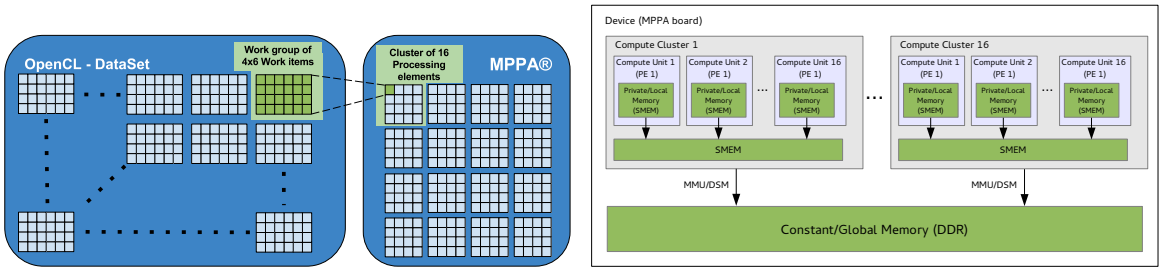


Figure 3.4 – *OpenCL Mapping of Applications and Memory Model*
Source: Kalray's OpenCL User Manual

Relation with the Nvidia CUDA Programming

Compute Unified Device Architecture (CUDA) is a proprietary API and a parallel computing platform for GPU SoC, designed by the Nvidia company, supporting C/C++ and Fortran. Nvidia is the world's first GPUs company. They invented the GPU in 1999, and have been a leader of parallel computing since that time. The CUDA programming API is close to OpenCL but less verbose as CUDA is dedicated to GPU programming; thus, less generic. CUDA abstracts the GPU programming by providing low-level virtual instructions to the Nvidia GPUs for tuning and optimizing the execution of kernels. Such

a level of application description makes it possible to reach the GPU's peak hardware throughput but at the cost of using a proprietary API. In any case, best performances of Nvidia GPUs are obtained using Nvidia optimized libraries like the cuBLAS, cuDNN, and cuFFT [Nvi18]. The vendors' level of optimizations are limited by the efficiency of the well-known parallelization techniques and its roofline model [WWP09] but also bottlenecks of the targeted architectures.

A Word on the Khronos Group

Founded in 2000, the Khronos Group is composed of universities and companies (large and small) like ARM, Intel, and Nvidia. Together they define new APIs, documents, file formats, and new open standards for the computing industry. The Khronos Group is royalty-free and aims to develop cross-platform technologies to help to solve the following problems: *3D graphics computing, virtual and augmented reality, parallel computing, neural networks, and vision processing* targeting computer desktops, embedded and safety-critical devices [Gro18]. The recent standards of the Khronos Group are the OpenVX API, the Vulkan API, the Neural Network Exchange Format (NNEF) and the SPIR-V Intermediate Language.

3.2.3 OpenACC & OpenMP 4.0 with Modern Compilers

OpenACC [WSTaM12] and OpenMP 4.0 [Ope13] are modern APIs for host-based offloading of computations onto an accelerator. OpenACC and OpenMP 4.0 aim to be at a high level of hardware abstraction compared to OpenCL or CUDA programming where the machine is exposed to the programmers; therefore, they OpenCL and CUDA are not easy to use (complicated). As such, OpenACC execution backends can be implemented using OpenCL or CUDA. OpenACC and OpenMP 4.0 propose compile time directives to gather code sections to be offloaded onto a heterogeneous accelerator. If no accelerators are specified or exist the parallelization is performed on the host. It is appreciated as it allows code portability in any case as in section 3.1.3. The memory can not only be shared with accelerators, but it can also be private to the accelerator. OpenACC and OpenMP 4.0 allow memory buffers to be mapped close to the computing resources.

OpenACC implements three parallelisms levels. The *Gangs* are used for coarse-grained, *Workers* are used for fine-grained and *Vectors* for Single Instruction, Multiple Data (SIMD) operations. The first experimental OpenACC implementation was released in GCC-5. Today the GCC-8 release implements OpenACC for Nvidia GPU out-of-the-box with a high level of maturity, directly mapped onto the CUDA API.

As an extension of multi-threading OpenMP 3.0 seen in section 3.1.3, OpenMP 4.0 adds the support of the *target* close to specifying where the scoped computations shall be offloaded. OpenMP 4.0 is available onto the POSIX runtime for SMP but also targeting Nvidia GPUs with modern compilers. The memory model is similar to OpenCL, with the host memory and the device memory. All data movements are handled by the host using either directly mapped memory or/and explicit Direct Memory Access (DMA) communications through system calls. In all cases, data movements are based on compiler directives for the offloading of data computations.

3.3 Dataflow Models

3.3.1 Introduction

Graphical, block-based or diagram representations of applications are intuitive to use for describing computer systems. The well-known Unified Modeling Language (UML) model [Exe02] is a decent example of a graphical model, which is used by many high-level designers for system engineering or the conception of object-oriented software systems. Most designs for automating simple systems are based onto Grafcet (Petri-net) [MHH+85] and Ladder [Win56] programming which are basic drawing-based programming models where the designers describe Finite-State Machines (FSMs) using *States* and *Transition Conditions* between these *States*. One of the first automated dataflow tools called BLOck DIagram compiler (BLODI)[KJLV61], was pioneered in 1961. From that time, BLODI provided primary predefined functions like *adders* and *multiplier* that can be connected to build a computer system. High-level commercial system designs also exist such as the Matlab Simulink®released for the first time in 1984 by Mathworks.

In this thesis, the focus will be put on the Synchronous Dataflow Graph (SDFG) model that is widely used for application description and inspired many other models. The Synchronous Dataflow Graph (SDFG) offers an interesting compromise between analyzability and expressiveness.

In the dataflow community, the dataflow programming models are also called Model of Computation (MoC). But in this thesis, we will call it a programming model for consistency.

Section 3.3.2 makes an overview of the dataflow programming model and presents one of the first model: the Kahn Process Network (KPN). The Dataflow Process Network (DPN) model is presented in Section 3.3.3 that defines the basics of dataflow models. Static and dynamic dataflow models are respectively presented in Section 3.3.4 and 3.3.5. The Section 3.3.6 presents a parametrized dataflow meta-model, that is used in the embedded reconfigurable dataflow runtime, namely SPIDER.

3.3.2 Dataflow Overview, the Kahn Process Network

Dataflow programming models are widely used for the specification of data-driven algorithms in many application areas. Dataflow programming models are architecture agnostic, which makes them highly valuable for the specification of applications that can be deployed on a wide variety of embedded systems.

As part of the first proposed dataflow programming models, the Kahn Process Network (KPN) was proved to be Turing complete [Gil74]. Such a property means that the model can compute anything that can be described by an algorithm but with the current computing physical limits (computer memory or processing time). Kahn Process Network (KPN) defines a network of potential concurrent tasks that are interconnected by directed unbounded First-In-First-Out queues (FIFOs). FIFOs gather data tokens to be consumed by the tasks when the production and consumption data of these tasks are available. In the KPN, by definition [Gil74], tasks and data tokens are indivisible.

3.3.3 Dataflow Process Network

The Dataflow Process Network (DPN) programming model was defined by Lee and Parks in [LP95] which is a generalization of the Kahn Process Network (KPN) programming model. The Dataflow Process Network (DPN) programming model is formally presented as follows:

Definition 3.3.3.1
A Dataflow Process Network (DPN) is a directed graph which is given by $\mathbf{G} = \langle \mathbf{V}, \mathbf{E} \rangle$ *where:*

- $\boldsymbol{V}$ *is the set of vertices of a graph* G*, where each vertex* $v \in V$ *represents an indivisible computational task, also called an* actor*, of the DPN. An actor is defined as follows:*
  - $\mathbf{d}_{input\_data}$ *refers to the set of input data ports of the actor* $v \in V$*.*
  - $\mathbf{d}_{output\_data}$ *refers to the set of output data ports of the actor* $v \in V$*.*
  - $\mathbf{FC} = \{FC_1, FC_2, ..., FC_n\}$ *is the set of Firing Conditions of the actor. When the Firing Conditions (FC) of an actor* $v \in V$ *are satisfied, the computation of actor* v *can be triggered. It is usually called the firing of an actor.*
  - **Rate** *refers to the number of indivisible data tokens consumed or produced on a given input data port or output data port respectively. The actor is executed when the number of tokens to consume for one firing of the actor is reached. The rate is non-deterministic and may depend on the internal state of the actor, on the number and value of tokens in FIFOs connected to the actor, or on time or randomness. In other words, the rate defines the number of data tokens of input and output ports of an actor of a dataflow graph.*
- $\mathbf{E}$ *represents a set of edges of graph* G*. Each edge* $e \in E$ *is an unbounded First-In-First-Out queue (FIFO) interconnecting two actors. A FIFO* $e \in E$ *connects a producer* p*, which writes data tokens in the FIFO, and a consumer* c *respectively connected to the source and sink ports of an actor* $v \in V$ *which reads the data tokens. A FIFO also implements a delay if any. The delay corresponds to the number of tokens placed in the associated FIFO at application initialization. The delay is usually used to represent recursive computations with DPN.*

As a summary and as a high-level definition, a dataflow graph is composed of communication *edges* representing FIFOs, that connect *vertices or actors* responsible for performing the computations. Figure 3.5 shows a DPN example and its semantic. In this chapter, the presented dataflow programming models are specialization in the Dataflow Process Network (DPN) model.

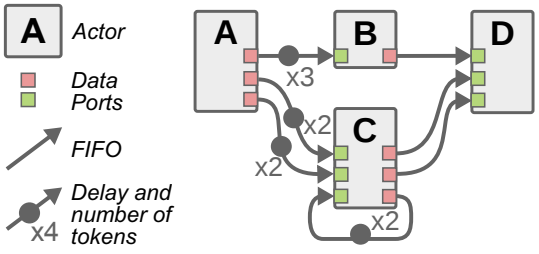


Figure 3.5 – *DPN Programming Model Example and Semantic*

3.3.4 Static Dataflow Models

Static dataflow programming models are not reconfigurable and deterministic. Therefore, such programming models have their sequence of firing rules of all actors composing the graph known at compile time. As such, it implies that the production and consumption rates of all actors are known at compile time.

An essential property of some dataflow programming models is the *data parallelism* (Section 2.1.1). The *data parallelism* of a dataflow graph is given by the Repetition Vector (RV) of actors within the graph. The Repetition Vector (RV) is formally explained as follows:

Definition 3.3.4.1 (Repetition Vector)
In dataflow programming model, the Repetition Vector (RV) refers to the number of execution of actors for single graph iteration. The RV defines the number of firings of each actor as a function of the production and consumption rates of FIFOs, so that, the dataflow graph is consistent and schedulable.

The RV of a Synchronous Dataflow (SDF) graph G is a vector containing an integer value $RV(a)$ for each actor a of G. An SDF graph completes a graph iteration when each actor is executed as many times as specified by the RV, thus bringing back the graph to its initial state in terms of the number of data tokens stored in each FIFO. The Repetition Vectors (RVs) are computed at compile time using static data rates of actors [LM87], or it can be computed at runtime in dynamic dataflow programming models if supported.

In this section, several static dataflow programming models are listed and some differences between them are highlighted.

- **Synchronous Dataflow**

  The SDF programming model introduced in [LM87] is a specialization of the DPN programming model that specifies for each FIFO the fixed number of data tokens produced and consumed at each execution (firing) of connected actors. SDF is probably the most studied dataflow programming model. The SDF programming model popularity is largely due to its **analyzability**, its predictability and its natural description of concurrency, which make it suitable for efficient execution on MPSoCs.

- **Single-Rate (SR) SDF, Homogeneous SDF or Directed Acyclic Graph (DAG)**

  Also known as the Single-Rate Directed Acyclic Graph (DAG) (SRDAG), the Single Rate SDF programming model is a specialization of the Synchronous Dataflow programming model where the number of consumed and produced data tokens is equal on each FIFO. More formally it means that $\forall e \in E, rate(prod(e), e) = rate(cons(e), e)$ for a given $G = \langle V, E \rangle$ dataflow graph. The graph is acyclic, meaning that no feedbacks or cycles are allowed. As already seen, the rate defines the number of data tokens of input and output ports of an actor of a dataflow graph. The Figure 3.6 shows an example of an SDF where its corresponding SRDAG transformation is given.

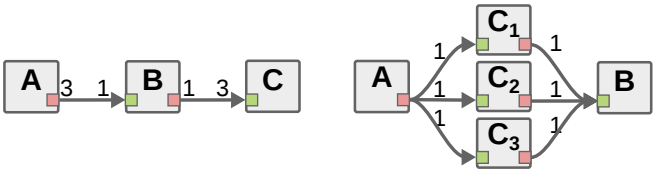


Figure 3.6 – *Single-Rate Transformation SDF (left) to Single-Rate DAG (SRDAG) (right)*

- **Cyclo-Static Dataflow (CSDF)**

  The Cyclo-Static Dataflow (CSDF) programming model generalizes the SDF programming model and provides statically the ability to vary the production and consumption rates of an actor $v \in V$ over graph iterations with a cyclic pattern. The

cyclo-static properties provide finer tuning of parallel application patterns, but it is more complex to use when application graphs are big because of graph consistency management. The reason for this complexity is that the translation from CSDF to SDF is exponential. Figure 3.7 shows a CSDF example where production and consumption rates of actor A varies over three graph iterations. The actor A produces 1, 2, and 3 data token triggering respectively 1, 2, and 3 times the B actor thanks to the RV.

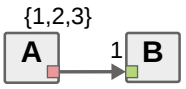


Figure 3.7 – *Cyclo-Static Dataflow (CSDF) Graph Example*

Hierarchical Dataflow Models

The hierarchy is an important feature for a dataflow programming model. Indeed the hierarchy in computer systems or in a dataflow application provides structuring and modularity. In dataflow models, the hierarchy mechanism associates an actor to a subgraph instead of code. We present diverse hierarchical dataflow programming models below.

- **Simple SDF Hierarchy: a Non-Compositional Dataflow Programming Model**

  The simple hierarchical SDF programming model, introduced in [LM87], adds the possibility to associate an actor to a Synchronous Dataflow (SDF) subgraph. The subgraph can contain several levels of hierarchy. The Flattening graph transformation of a hierarchical dataflow graph consists in replacing the hierarchical actor with their corresponding subgraph. However, an issue of the simple hierarchy programming model is that the compositionality of hierarchical actors is not guaranteed between the subgraphs and top graphs. In Figure 3.8, on the left can be seen a hierarchical dataflow graph. To be concise, in a subgraph, when the internal RVs of actors imply different production and consumption rates than the ones of the data input and output data ports of the enclosing hierarchical actor, the compositionality rule of the simple SDF hierarchy programming model is violated as seen in Figure 3.8 on the SRDAG.

  The compositionality of a dataflow programming model is defined as the behavioral independence of the internal specification of the actors of a dataflow graph as presented in [TBG+13]. A compositional dataflow programming model implies that modifications to a subgraph of a hierarchical graph will not influence the consistency or schedulability of this hierarchical dataflow graph.

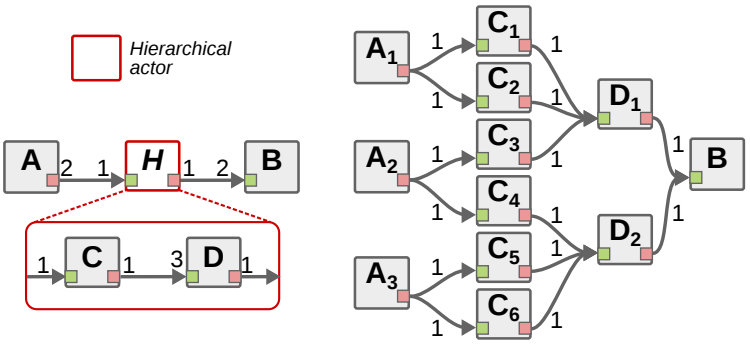


Figure 3.8 – *Flattening and the Single-Rate Transformation Hierarchical SDF (left) to Single-Rate DAG (SRDAG) (right)*

- **Interface-Based SDF Programming Model**

  The Interface-Based SDF (IBSDF) [PBR09] programming model is a hierarchical extension of the SDF model. In addition to the SDF semantics, Interface-Based SDF (IBSDF) adds the possibility to specify the internal behavior of an actor with a dataflow subgraph instead of specifying it with code (compared to simple actors). In the IBSDF programming model, the compositionality is enforced by the model semantics and execution rules, which make it possible to translate each hierarchical actor into an equivalent code with fixed production and consumption rates.

  Contrary to the simple SDF hierarchy, the IBSDF programming model ensures the compositionality of hierarchical dataflow graphs. As seen in Figure 3.9, the IBSDF adds interfaces at the edges of hierarchical actors. An input interface has a broadcast role, called *Brd* in Figure 3.9 which produces several times the same data token at the input of the hierarchal actor. An output interface has a round buffer role which sends the last data token in the output of the hierarchical actor. In IBSDF, the compositionality feature enables independent computations of the RV of each hierarchical graph or subgraph [PBR09] (at multiple levels). The IBSDF programming model is a compositional dataflow programming model regarding the parent graphs and children graphs.

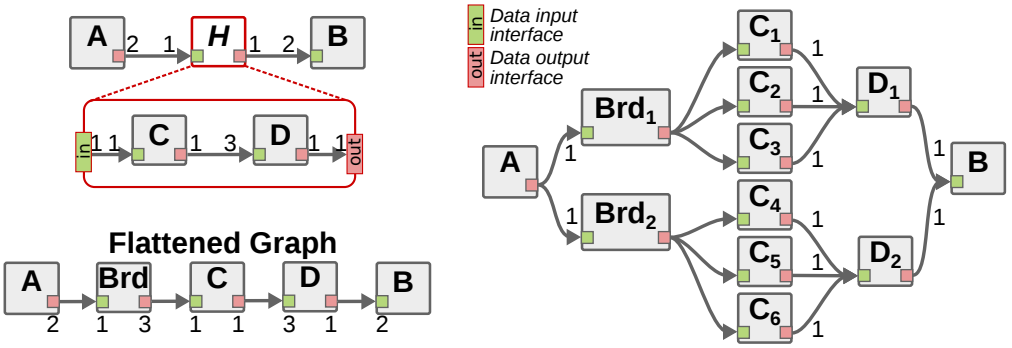


Figure 3.9 – *Flattening and Single-Rate Transformation of IBSDF (left) to Single-Rate DAG (SRDAG) (right)*

- **Deterministic SDF with Shared FIFOs**

  Another hierarchical generalization of the SDF programming model called Deterministic SDF with Shared FIFOs (DSSF) is proposed in [TBG+13]. The main difference

between the DSSF and the IBSDF programming models is that DSSF compositionality results from a graph analysis, whereas IBSDF graphs are inherently compositional. In DSSF, a bottom-up analysis is used to expose compositionality of the hierarchical graph, when possible. Based on this analysis, a hierarchical actor can be translated into equivalent modular code with variable consumption and production rates.

Consistency and Schedulability

The consistency of a static dataflow graph noted $G = \langle V, E \rangle$ is usually checked using the topological matrix of the graph. The topological matrix T is a size of $|V| * |E|$ in which rows represent *edges* (E), and columns represent *vertices* (V). A coefficient of $T(i, j)$ of this topological matrix is computed as the number of tokens produced ($+n$) and consumed (-1) for each edge related to each vertex. Lee proved in [LM87] that the graph G could be scheduled, if and only if the rank of the T matrix if less than the number of vertices in graph G.

3.3.5 Dynamic Dataflow Models

Reconfigurable dataflow programming models offer a tradeoff between dynamicity and predictability that can be exploited by a runtime manager to verify application properties or to perform optimizations at runtime, like the mapping of actor computations [HPD+14]. In real life, the implementation of dynamic dataflow programming models is a difficult task to make it scale efficiently because of the sequentiality of the control path that is managed by software.

- **Parametrized SDF**

  Bhattacharya and Bhattacharyya introduced the Parameterized SDF (PSDF) in [BB01]. The PSDF programming model inherits the SDF programming model properties, adds the hierarchy and adds parameters that can be used to change the production and consumption rates of edges of actors; thus, changing the RV of actors composing the graph. The PSDF is a dataflow meta-model, meaning that the semantics of existing dataflow models can be augmented with a semantic element from the meta-models. For instance, the model makes the reconfiguration of hierarchical actors (subgraphs) possible at runtime. The parameter is an integer value that can be modified during the execution of the graph at runtime by other actors; thus, the mapping and scheduling might be impacted. The dynamic value of the parameters can also be bounded. Such parameters imply quasi-static schedules. The Boolean Parametric DataFlow (BPDF) [BFGL13] is also essential and offers more reconfigurability than the PSDF. The hierarchy semantics in the PSDF programming model is slightly different from the semantic implemented in the IBSDF programming model. A PSDF graph performs a graph initialization process that is triggered at each graph iteration, and it configures the production and consumption rates of the ports of actors. A hierarchical actor implements an additional sub-initialization process (executed before the initialization process) that consumes the input data token of one firing of the hierarchical actor and finalizes its configuration.

- **Other Dynamic Dataflow Programming Models**

  Other dynamic dataflow programming models exist like the Schedulable Parametric Dataflow (SPDF) programming model explained in [FGP12], the Scenario-Aware Dataflow (SADF) programming model described in [TGB+06] and the Boolean DataFlow

(BDF) programming model presented in [BL93]. All of these dynamic models extend from the Synchronous Dataflow (SDF) programming model. Scenario-Aware Dataflow (SADF) provides analyzability, and a scenario gives the production and consumption rates on the ports of actors. However, the scenario of some particular actors, called detectors, (explained in the model in [TGB+06]) is defined by a stochastic approach (Markov chain) that is difficult to use in real life applications [Des14]. Schedulable Parametric Dataflow (SPDF), known as a Turing-complete model, gives decent analyzability and predictability. SPDF makes it possible to modify a parameter of an actor by a modifier actor. The modification of a parameter influence the production and consumption rates of an actor, thus its schedule.

3.3.6 Parametrized Interfaced-based SDF

The Parameterized and Interfaced SDF (PiSDF) programming model is a reconfigurable dataflow programming model presented by Desnos et al. in [DPN+13]. The Parameterized and Interfaced SDF (PiSDF) inherits its hierarchy semantics from the IBSDF programming model introduced in [PBR09].

The PiSDF programming model extends the SDF programming model by adding hierarchical interfaces. Such an interface can implement a set of parameters that are associated with vertices to make graph configuration possible, and dependency parameters for propagating information of elements of the graph between each other. Also, it must be noted that an interface of a hierarchical actor has the same property of a round buffer. The reconfiguration in the PiSDF programming model is based on parameters which can modify the rate of a graph. These production and consumption rates of actors can be specified with expressions depending on these parameters. In PiSDF both static and dynamic parameters can be specified, allowing partial graph reconfiguration. Following PiSDF execution rules [DPN+13], an actor may trigger reconfiguration of the graph topology and intrinsic parallelism by setting a new parameter value at runtime.

From Desnos et al., the Figure 3.10 summarizes the semantics of the Parameterized and Interfaced dataflow Meta-Model (PiMM) model and comparison with previously initiated programming models.

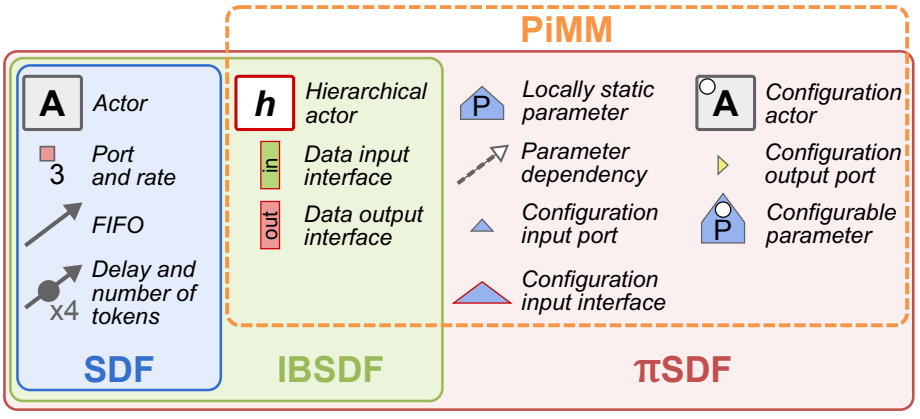
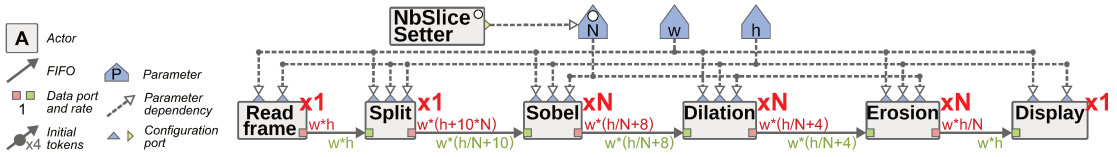


Figure 3.10 – *PiMM Semantics (Source [Des14])*

Figure 3.11 depicts the graphical elements of the PiSDF semantics and gives an example of a graph implementing a video filtering algorithm. At each iteration of the graph, which corresponds to the processing of a new frame, the *SetNbSlice* actor triggers a reconfiguration of the data rates by assigning a new value to parameter N. Reconfigurations enable a

dynamic variation of the number of parallel executions of the *Sobel*, *Dilation*, and *Erosion* actors.



(a) *PiSDF programming model semantics*

(b) *PiSDF graph of an image filtering application*

Figure 3.11 – *PiSDF Programming Model Semantics and Example*

3.4 Graph Scheduling and Memory Allocation

In this section, we explain the problem of graph scheduling and memory allocation related to dataflow-based programming. Such issues have been widely studied in the literature and are not the purpose of this thesis. However, it is required to mention and understand them as some techniques are directly used in the contribution part.

3.4.1 Scheduling Methods for SDF Graphs

The mapping and scheduling of a dataflow graph onto an embedded multi/manycore systems is not a trivial task. When the application has been described and split into pieces using a dataflow programming model, the problem consists in assigning pieces of the dataflow application to the available computing resources. The goal of the mapping and scheduling is usually to maximize the throughput of the application or minimize the latency of the application, by trying to exploit the parallel architecture as efficiently as possible.

The mapping and scheduling problem is known to be NP-complete, as shown by [BB07]. As such the time to map actors (pieces of an application) onto computing resources theoretically increases exponentially with the number of application pieces to map and the number of computing resources. However, several mapping and scheduling algorithms can be found in the literature as presented in [BB07, BL93, Pap16, PNA10, Kwo97]. Many of the proposed solutions make it possible to solve the mapping and scheduling problem in a polynomial amount of time, but they do not result in an optimal solution.

Multi-core Scheduling

Defined by Lee et al. in [LH89] and [BL93], the scheduling of a dataflow application; that is split into pieces, is divided into the following steps:

- The **Extraction** defines the retrieving of parallelism within a given application description.
- The **Mapping** consists in assigning each task and subtask (finer-grained) to a specific Processing Element (PE) of the targeted machine. The mapping is usually under constraints that are either user-defined or provided by the architectural features such as the PE efficiency, the communication latency, and throughput or the power consumption.

- The **Ordering** is the sequence of tasks to be fired (sequentially) for each PE of the targeted machine.

- The **Timing** phase consists in assigning the tasks to a start time within the threads where the tasks were previously mapped. Sometimes the timing phase is set to best effort, meaning that the tasks are scheduled by an OS when the input data of the task are available, and enough place is available in the output data buffer.

Fundamental Algorithms

The Depth-First Search (DFS) algorithm is a base of most graph parsing, analysis, transformation algorithms. Introduced in 1972 by Tarjan [Tar72], the Depth-First Search (DFS) algorithm has been used to find connected components. A DFS algorithm operates linearly in time; thus, has a complexity of $O(n)$, where n is the number of vertices. Therefore, DFSs are intensively used in compilation passes.

Based on a Depth-First Search (DFS) algorithm, the topological sort gives the data dependency schedule of a graph [KL95]. The topological sort places vertices of a graph to be scheduled in a list, ordered with respect to the directed edges representing the data dependencies. As the topological sort is based on a DFS algorithm, the topological sort algorithm operates in linear time. In this thesis, we apply topological sorts to SRDAGs, formally noted as $G(V, E)$, to deal with the dependency order. The pseudo code of the topological sort is given in algorithm 1.

Algorithm 1 Topological Sort Algorithm.

```
1:  Input: Set_of_Vertices { Vertices composing a Graph } of a DAG
2:  Output: Set_of_List_of_Sorted_Vertices { { } }
3:  while Set_of_Vertices Not Sorted do
4:    List_of_Current_Sorted_Vertices = { }
5:    for Current_Vertex in Set_of_Vertices do
6:      if Current_Vertex is in the Set_of_List_of_Sorted_Vertices then
7:        continue
8:      end if
9:      Set_of_Predecessors = Get_Predecessors_Of_Vertex(Current_Vertex)
10:     if Set_of_Predecessors are Sorted or Empty then
11:       Add Current_Vertex to List_of_Current_Sorted_Vertices
12:       Mark Current_Vertex as Sorted
13:     end if
14:   end for
15:   Add the List_of_Current_Sorted_Vertices to Set_of_List_of_Sorted_Vertices
16: end while
```

Once the topological sort is applied to a Directed Acyclic Graph (DAG), many scheduler algorithms exist which are classified as static or dynamic. A summary of static scheduling algorithms is made in [SAS15]. For dynamic scheduling of DAGs, we usually use the LIST scheduler explained by Brucker in [BB07]. This thesis reuses such algorithms and adapts them to fit the targeted problems of automating the mapping and scheduling of parallel dataflow-based application onto clustered manycore architectures.

3.4.2 Memory Allocation

The memory allocation is an essential procedure for efficient execution of parallel applications when targeting complex memory hierarchy architectures as seen in section 2.2.1. Multiple levels of cache or local memories in the memory hierarchy of computer architectures make it challenging to deploy automatically parallel applications. Bad placement of buffers in the memory hierarchy, will lead to misuse of the Processing Elements (PEs) composing the computer system.

When allocating memory resources for the execution of a dataflow graph, the misuse of PEs is mostly due to the lack of data locality, meaning that PEs will spend a high amount of time in memory access dependency stalls. That is why a smart and efficient mapping/scheduling and memory allocation are required to use as efficiently as possible the memory that is close to the PEs.

Memory Allocation Methods

The memory allocation consists in assigning a memory buffer (start address and size usually in bytes) in a continuous virtual memory address space. The memory buffer lifetime is given by the differences between the first and last memory access timestamp in the scheduled application. In both static [ALP97] and dynamic [BAMJ13] dataflow applications, the memory needs to be managed. For optimization purpose, the memory consumption has to be minimized (mainly depends on the scheduling) and temporal and spatial data locality needs to be maximized. The minimization of the memory footprint of an application has been proved to be an NP-hard problem by Bouchard et al. in [BČH09].

The memory allocation has been widely studied in the past decades [Joh73]; however, a decent memory allocation mainly depends on a memory aware schedule [BL93], the buffer sizing technique [DPNA15] [SGB06], and graph level memory optimization [DPNA15] [Des14].

Online Memory Allocation Algorithms

In the literature, many memory allocator algorithms have been designed for applications running in real-time (online memory allocation). The allocation of memory buffers is done in the *heap* as seen in Section 2.4.2.

- **First-Fit (FF)** was introduced by Johnson [Joh73] in 1973, and it is usually implemented using double linked lists. The First-Fit (FF) memory allocator is a sequential fit algorithm which returns the first buffer address with a given size that is available within a memory space.
- **Best-Fit (BF)** [Joh73], is similar to the FF. However, it allocates memory buffers that fit the best area in a memory space. More precisely, the used heuristic tries to minimize the lost memory in the memory space.
- **Binary-Buddy** is probably the oldest memory allocator (1965). The algorithm splits the memory space into static equal pieces (usually a power of 2) and attempts to return memory buffers that best fit in this memory space.
- **Doug Lea**, known as the *dlmalloc* [LG96], is considered to be the *best* existing allocator for a general purpose system. The *dlmalloc* is one of the most used memory allocator providing good performances on a wide range of applications. The *dlmalloc* is available in the GNU Compiler Collection (GCC) project in the C and C++

runtime libraries [SLwRMSD18]. The Doug Lea allocator is a refinement of the Best-Fit (BF) with the binning of sizes of memory chunks.

- **Half-fit**, explained in [Oga95], is similar to the **Binary-Buddy** but provides a short Worst-Case Execution Time (WCET) as it has only few memory accesses to operate; thus, it drastically reduces Translation Lookaside Buffer (TLB) (Section 2.4.3), and data cache (Section 2.2.1) misses.

Offline Memory Allocation

Offline memory allocations are used when the schedule of an application can be static. Reaching optimal memory minimization of a statically scheduled application is more straightforward than dynamic memory allocation of an application. Indeed, offline allocators have a global knowledge of the mapped application; however, it is still not a trivial task. An offline allocation provides better optimization opportunities regarding the placement of buffers for data locality and also memory footprint minimization, but solving this problem is a challenge. The literature shows offline allocators in papers [MB00] [DGCDM97] [Des14] [BČH09] that solve the global problem of allocating memory buffers of statically scheduled tasks. They use graph coloring techniques modeling exclusion graphs to understand non-overlapping lifetime buffers of the tasks to identify memory reuse opportunities. They also use online allocators for offline memory allocations by simulating the static schedule. The simulation of the static schedule aims to provide the lifetime of each buffer of the schedule application. Memory buffers are allocated at their first usage, and they are freed (recycled) when the last task using them completes. Found addresses of memory buffers are then saved either in the *.data* or *.text* section of the Executable and Linkable Format (ELF) file. The CPUs just read addresses of these memory buffers to proceed with the computation.

3.5 Rapid Prototyping and Existing Dataflow-based Tools

The motivation of rapid prototyping is to bridge the gap between the ever-increasing hardware complexity and the engineer's productivity. As explained in [CH89], rapid prototyping operates using system and application models with a specific semantic that is then used to generate *ready-to-use* simulations or prototypes automatically. Using an application specification written by an engineer, the rapid prototyping tool aims to abstract the implementation which is left to the automatic tool. Rapid prototyping tools usually operate with high-level programming models or Domain Specific Language (DSL). A DSL is a computer language that has been designed by a group of experts for a particular type of applications.

3.5.1 PREESM: an Open Source Rapid Prototyping Framework

The PREESM is an open source framework based on Eclipse developed by the Institute of Electronics and Telecommunications of Rennes (IETR). PREESM allows the developer to design dataflow-based algorithms using the PiSDF programming model or the IBSDF programming model. The developer focuses on the dataflow application description and PREESM generates code for the targeted embedded MPSoCs. The PREESM project[1] has been developed for research, development, and education purposes.

[1] Available at `https://github.com/preesm/preesm`

The PREESM's development workflow presented in Figure 3.12 shows typical design and compilation steps from the IBSDF graph specification by the developer, using a graphical user interface, to the software synthesis.

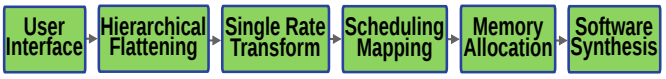


Figure 3.12 – *Typical PREESM's Rapid Prototyping Workflow*

Each step of the workflow is described as follows.

- **User Interface.** The user interface is a *point-and-click* graph edition interface. It allows the developer to design IBSDF or PiSDF application graphs. The vertices of the model are then linked to C/C++ functions. Once the graph is completed the user gives a scenario and the top level application graph entity. The scenario describes the targeted architecture and its properties like the speed of the memory, the size of data type, the number of cores and how they are interconnected.

- **Hierarchical Flatenning.** The hierarchical flattener is given a depth level to flatten the hierarchy. Flattening the hierarchy consists in replacing hierarchical actors with their equivalent sub-graph and connecting them to the parent graph. As such the flattening depth defines the granularity of the hierarchical flattening operation.

- **Single-Rate (SR) and Directed Acyclic Graph (DAG) Transformation.** The purpose of the Single-Rate (SR) transformation is to expose all the implicit data parallelism of the dataflow graph. During this process, a lot of actors can be generated depending on the Repetition Vector (RV) of actors in the transformed dataflow graph. Also, the transformation to a DAG is performed to simplify the mapping and scheduling process as each vertex has to be scheduled only once.

- **Scheduling and Mapping.** The scheduling and mapping are done statically, operating on a Directed Acyclic Graph (DAG), several mapping and scheduling strategies are available such as the FAST and LIST schedulers explained in [Kwo97]. Moreover, the scheduler uses the Architecture Benchmark Computer (ABC) framework, designed and implemented by Pelcat et al. [PMAN09], that gives the developer the opportunity to find the best tradeoff between accuracy and speed of the mapping and scheduling simulation.

- **Memory Allocation.** The memory allocation, designed and implemented by Desnos [Des14], is placed post-scheduling using refinements of the First-Fit (FF), Best-Fit (BF) and graph coloring memory allocator algorithms. The memory allocation either supports shared memory, where all buffers are allocated in a shared memory space, or distributed memories where buffers can be allocated to different memory pools. The distributed memory allocation is presented in [DPNA16].

- **Software Synthesis.** The current software synthesis back-ends of PREESM supports off-the-shelf multi-core processors running Linux or Windows, the Texas Instrument C6x embedded MPSoC [TIC13], and the heterogeneous Texas Instruments OMAP4 multi-core platform [HDN+12]. The software synthesis consists of generating C/C++ files containing: calls to functions implementing the internal behavior of actor, calls to communication primitives between the computing resources of the targeted platform and the synchronization. Usually, the generated files are then

integrated inside the pre-written platform specific project, designed to deploy the computations.

In PREESM, all steps of the workflow are static; thus, they are executed offline. In the next Section 3.5.2, we present a runtime that performs all of these steps online (except for the software synthesis).

3.5.2 SPIDER: an Embedded Reconfigurable Dataflow Runtime

The SPIDER embedded runtime was initially introduced in [HPD+14] as a runtime manager for the execution of reconfigurable PiSDF graphs on heterogeneous MPSoCs. The SPIDER runtime can also be used as a rapid prototyping tool to deploy easily reconfigurable dataflow applications described in PiSDF.

Overview of the SPIDER Runtime

Written in C++[2], the SPIDER runtime is organized as follows. SPIDER takes as an input a PiSDF graph that is developed in the PREESM framework which is presented in Section 3.5.1. The SPIDER runtime then executes the graph transformations (flattening and SR-DAG) if all parameters are available. If not, these transformations are performed when reconfigurable parameters are set during the processing of the graph. Then, the SPIDER runtime performs the mapping and scheduling and sends computation commands to the PE of the platform.

SPIDER implements the *master/slave* organization [SSKH13]. The master of the runtime system is called the SPIDER Global RunTime (GRT) which manages the PiSDF graph topology and takes mapping and scheduling decisions. The slaves, called the Local RunTimes (LRTs), are mapped onto the computing resources of the platform in charge of executing the computation of actors. Local RunTimes (LRTs) are general purpose processors, specialized processors or custom accelerators. As such, LRTs are lightweight slave processes that execute actors. Moreover, the SPIDER GRT can perform the computation of actors. Indeed, the SPIDER GRT is usually placed on a general purpose core.

Spider Structure

The internal structure and behavior of the SPIDER runtime are shown in Figure 3.13. The runtime implements the master/slave model and uses job queues to transmit control commands between the SPIDER GRT and the LRTs. Indeed, as the SPIDER GRT is responsible for the scheduling and the management of the memory, low latency communications are required to control the slave LRTs. Two kinds of queue exist and they are described as follows:

- Data channels that are used for the data path (high-throughput) of data tokens exchange. Such data tokens are the memory buffers where the computation of actors is carried out.

- Control queues that are used for the sending of computation commands, reconfiguration parameters (partial or global) and the profiling of the dynamically scheduled reconfigurable dataflow graph.

[2]available at `https://github.com/preesm/spider`

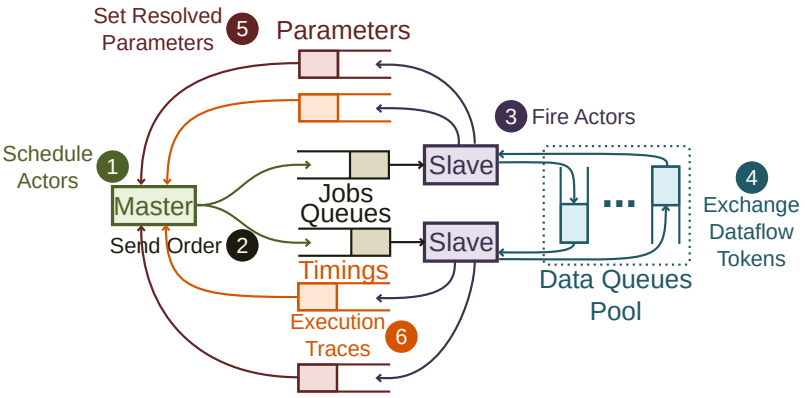


Figure 3.13 – *Internal Structure of the SPIDER Runtime*

SPIDER Operations

The execution steps followed by SPIDER to run an actor are numbered in Figure 3.13. First (Step 1), the GRT schedules an actor on a PE of the architecture, and sends (Step 2) the execution order through the dedicated *job queue* of the LRT of this PE. A *job* is a message that embeds all data required to execute one instance of an actor: a job ID, location of actor data and code, and identifiers of preceding actors in a graph execution. When an LRT starts an actor execution (Step 3), it waits for data tokens to be available in the input FIFOs specified in the job message, among a pool of data FIFOs. On actor completion, data tokens are written in output FIFOs (Step 4), and the LRT sends new parameter values (Step 5) if any, and execution traces back to the GRT for reconfiguration, monitoring and debugging purposes (Step 6). Each LRT is associated with a *job counter* that stores the integer job ID of the last executed job. As the job IDs increase monotonically both with scheduling order and data dependencies between jobs, these job counters can be used for synchronization purposes between LRTs, to check whether an LRT already executed a given job.

Support of SPIDER for MPSoC

The SPIDER runtime divided into key parts that are either generic or platform specific. The SPIDER runtime is divided as follows.

- The generic SPIDER GRT runtime is in charge of the management of the scheduling.
- The generic LRT is in charge of the computation.
- The platform-specific communications and synchronizations for the LRTs.
- The platform-specific communications and synchronizations for the SPIDER GRT.
- The platform-specific management of the parallelization (machine specific multitasking).

Currently, open-source implementations of the SPIDER runtime have been proposed for general purpose x86 architectures which are either based on Linux or Windows. Also for embedded MPSoCs, the SPIDER runtime has been implemented for both the Texas Instruments (TI) Keystone Digital Signal Processor (DSP) processor architectures, and

Xilinx's Zynq heterogeneous platforms [HPD+14]. In this thesis, we will attempt to show the feasibility of running such embedded reconfigurable dataflow runtime on the Kalray MPPA® processor.

3.5.3 Other Tools Based on Dataflow Programming Models or Languages

In the literature, several dataflow-based tools are found that are mainly designed by researchers or companies. All of these tools have their own semantics, but they target the same objective: providing the developers with a higher level of abstraction using a dataflow programming model in order to simplify the programming of MPSoCs with the desired application requirements (real-time, Worst-Case Execution Time (WCET), performance latency and throughput).

Orcc is a compilation framework based on a language called RVC-CAL. This language is based on the DPN programming model which is a dynamic and non-deterministic dataflow model [CPG+]. Unfortunately, this language does not provide the analyzability capabilities required for the programming of embedded systems.

Scade/Lustre is a language for reactive system programming as it provides a logical time notion. Scade/Lustre is described in [Ber07], and a code generator has been written in Python for targeting the MPPA® processor in [GMRdD18]. The main difference with the other approaches is that SCADE allows sampling (sensors typically) in the firing of nodes.

SigmaC is a language based on the CSDF model which is an extension of the SDF model. SigmaC [dDAB+13] was supported in the Kalray MPPA® toolchain and was well-suited for time-critical applications with a static behavior, i.e., computations are the same for all the data and cannot change dynamically.

SLX Dataflow is the proprietary dataflow framework of the Silexica company [3]. Like the open-source framework PREESM, the Silexica framework [CLA13, CCS+08] provides automatic optimization of dataflow applications. The framework implements the modeling of the targeting architecture, static and dynamic code source analysis, memory communication optimizations and chooses the best available runtime environments.

3.6 Conclusion

This chapter introduces parallel programming models that are either inspired on, developed, enhanced or used in the contribution part of this thesis for the Kalray MPPA® processor. We gave generalities about parallelism and how to interpret scalability and speedups of parallel implementations. These notions are used in all contribution chapters of this thesis.

The multitasking and multi-threading models for SMP machines are introduced. Pthread and the OpenMP multi-threading are presented in details in order to help the comprehension of the contribution of Chapter 6 that implements at bare-hypervised level an optimized multi-threading runtime for managing the PEs of the Compute Clusters (CCs) of

[3] https://www.silexica.com

MPPA® (but also the Input/Output Subsystem (IO)). We also explain the purpose of acceleration programming models with standard models like OpenCL, OpenAMP and OpenMP 4.0. The Kalray MPPA® processor can be used in acceleration using OpenCL offloading either from an x86 processor over the Peripheral Component Interconnect Express (PCIE) bus (x86 offload on MPPA®) or from a Linux running on the IO multi-core over of the Network on Chip (NoC) (stand-alone OpenCL). Such acceleration model is essential to be understood as Section 5.6.2 contributes to the Kalray OpenCL runtime and Section 9.2 implements a low-level offloading runtime from the IO multi-core to the Compute Clusters (CCs) at bare-hypervised level.

We then describe and define the state-of-the-art dataflow models. As seen, dataflow models are suitable to describe computing pipeline and express application parallelisms. The dataflow developer expresses the application in a dataflow model, and the presented rapid prototyping tools make the deployment on available computing resources of the targeting architecture automatically. Dataflow models are explained in details starting from the KPN to the latest dataflow models like the PiMM or the IBSDF model; both exploited in the contribution part of this thesis. Important features of the IBSDF dataflow model are the hierarchy and the RV that are carefully exploited in the contribution of Chapter 7. Moreover, the PiMM model, that can be expressed in the PREESM interface is the model used in the SPIDER embedded runtime. In Chapter 8, we propose an implementation of such an embedded reconfigurable dataflow runtime on a manycore machine like MPPA®. Such an implementation of a reconfigurable dataflow runtime is the first ever made to the best of our knowledge at this time (targeting an embedded manycore processor).

CHAPTER 4

Communication Protocols & Memory Consistency

This chapter presents diverse communication protocols commonly used in embedded and High-Performance Computing (HPC) systems. A particular focus is put on HPC communications and synchronizations on both hardware and software. The memory consistency is also explained to understand the diverse contributions proposed in this thesis.

As the new generation of clustered manycores aims to be used for embedded and HPC systems, they include hundreds of cores with shared local memories. They are programmed like super-computers, but it is a single chip. All manycores need to move data and synchronize cores efficiently to reach their peak performance and efficiency targets. In this thesis, the targeted experimental architecture is the Kalray Multi-Purpose Processor Array (MPPA)® [SEU+15] processor, which integrates a local memory, shared by the cores of each Compute Cluster (CC). For instance, it exists the Adapteva Epiphany 64 [VEMR14] and the Epiphany-V [Olo16], which integrate local memories attached to each core. Using these new architectures implies more complexity in the software to communicate between cores. However, it is a way to reach energy-efficient computing. This background chapter provides commonly used techniques for communications and gives advantages and drawbacks of them.

Section 4.1 is a background of several technologies for communication and synchronization that have been studied and analyzed to choose a subset of mechanisms for the contribution presented in Chapter 5. Then the two-sided and one-sided protocols are explained in details in Sections 4.2 and 4.3. Then, we detail fundamental notions on the memory consistency and coherence on parallel computer architectures in Section 4.4. The management of the memory consistency of the Kalray Very Long Instruction Word (VLIW) core is detailed in Section 4.5. The streaming and atomic memory accesses are also detailed, as they are widely used in Chapters 5 and 6.

4.1 State-of-the-Art of Communication Technologies for HPC

Section 4.1.1 is an overview of HPC hardware interconnects that have been designed by hardware manufacturers in the past decades. The Section 4.1.2 introduces HPC software programming Application Programming Interfaces (APIs) proposed by academics and vendors. The Section 4.1.3 presents standard OpenCL primitives, used for the performance optimizations of communications.

4.1.1 HPC Hardware Interconnects

This section explains existing technologies that are deployed in production by hardware vendors for high-performance data communications on well-known interconnects. Here we introduce the Infiniband technology and derived versions of it over Ethernet networks, as well as the latest communication technologies of Intel and Nvidia.

The Infiniband technology designed by Mellanox [RoC15] is widely deployed in high-performance systems and data centers. It natively supports Remote Direct Memory Access (RDMA) *Put-Get*, Send/Receive *read-write* and atomic operations. Based on the earlier Virtual Interface Architecture (VIA), the Infiniband specification only lists Verbs, that is, functions that must exist but whose syntax is left to vendors.

Vendors are free to create their own Verbs APIs which led to the Open-Fabrics Association (OFA) Verbs [CTK+09]. OFA Verbs have support for: two-sided and one-sided operations, always asynchronous; reliable and unreliable modes, connection-oriented and connection-less; remote direct memory access, send and receive; and atomic operations on remote memory regions. To allow the direct access to endpoint memory, this virtual memory must be pinned in physical memory and registered into the network interface I/O Memory Management Unit (MMU). OFA Verbs offer cross-platform support across Infiniband on IB network, iWARP on IP network and RDMA over Converged Ethernet (RoCE) on Ethernet fabric.

iWARP uses IETF defined Remote Direct Data Placement (RDDP) to deliver RDMA services over standard, unmodified IP network and standard TCP/IP Ethernet services. Enhancements to the Ethernet data link layer enabled the application of advanced RDMA services over the IEEE Data Center Bridging (DCB), that is, lossless Ethernet. In early 2010, this technology, now known as RoCE was standardized by the Infiniband Trade Association (IBTA). RoCE utilizes advances in Ethernet (DCB) to eliminate the need to modify and optimize iWARP to run efficiently over DCB. RoCE focuses on server-to-server and server-to-storage networks, delivering the lowest latency and jitter and enabling more straightforward software and hardware implementations. RoCE supports the OFA Verbs interface seamlessly.

The GPUDirect specification was developed together by Mellanox and Nvidia. It is composed of a new interface (API) within the Tesla Graphics Processing Unit (GPU) driver, a new interface within the Mellanox Infiniband drivers, and a Linux kernel modification to support direct communication between drivers. GPUDirect allows RDMA capable devices to direct access GPU device memories, so that data can be directly transferred between two GPUs without buffering in host memory. GPUDirect Verbs provide extended memory registration functions to support GPU buffer and GPU memory de-allocation call-back for efficient Message Passing Interface (MPI) implementations.

The Intel® Omni-Path technology competes with Infiniband, with the advantage that the interfaces can be integrated into the Intel® processor themselves. It can be used through the OpenFabrics library, which has an implementation of the Infiniband Verbs API as standardized by the OFA.

4.1.2 HPC Software Programming

Today's HPC programming models are based on the Single Program, Multiple Data (SPMD) execution model, where a single program is spawned on N processing nodes. There is one process per node, and each process is assigned a unique *rank* $\in [0, N]$. The main HPC programming model is the Message Passing Interface (MPI), which combines the SPMD execution model, explicit *Send/Receive* of data, and collective operations. The MPI stan-

dard introduced one-sided communications in MPI-2, which have been reworked and can be combined with split-phase synchronization in MPI-3.

Whereas most HPC applications still rely on message-passing semantics using traditional message-passing, the underlying communication systems have evolved several years ago to build on one-sided communication semantics, starting with Cray SHMEM library [Fei95][GL04]. The rise of Partitioned-Global-Address-Space (PGAS) languages like Co-Array Fortran [NR98] [MCASJ09], UPC and of Global Arrays motivated the development of one-sided communication layers, notably GasNet from Berkeley and Aggregate Remote Memory Copy Interface (ARMCI) from PNNL. Partitioned-Global-Address-Space (PGAS) languages and Global Arrays combine the SPMD execution model, one-sided communications, and collective operations.

The Cray SHMEM (SHared MEMory) library [CCP+10] was initially introduced by Cray Research for low-level programming on the Cray T3D and T3E massively parallel processors [Fei95]. This library defines symmetric variables as those with the same size, type, and address relative to the processor local address space, and these naturally appear as a by-product of the SPMD execution model. Dynamic memory allocation of symmetric variables is supported with a `shmalloc()` operation. Static data and heap data obtained through this symmetric allocator are implicitly registered. Thanks to the symmetric variables, it is possible to use one-sided operations easily such as *Put* and *Get* by referring to local objects only. *Put* and *Get* operations are explained in details in Section 4.3. Besides *Put* and *Get* variants, the SHMEM library supports remote atomic operations, and collective operations. The SHMEM library motivated the design of the F−− language [GNP90], one of the first Partitioned-Global-Address-Space (PGAS) languages, which evolved into Co-Array Fortran.

The Aggregate Remote Memory Copy Interface (ARMCI) [NC99] was designed as an improvement over Cray SHMEM and IBM LAPI (IBM SP) and is used as the base of the Global Arrays toolkit. The API is structured in three classes of operations:

- Data transfer operations: *Put*, *Get*, and *Accumulate*
- Synchronization operations: Atomic read-modify-write, and lock/mutex
- Utility operations: Memory allocation/deallocation, local/global *Fence*, and error handling

The Berkeley Global Address Space Networking (GASNet) library [Bon08] is designed as a compiler runtime library for the PGAS languages UPC and Titanium. It also provides the foundations for the Rice University Co-Array Fortran 2.0, which aims to correct a number of identified shortcomings [MCASJ09]. The GASNet library is structured with a core API and an extended API. The core API includes memory registration primitives and is otherwise based on the active message paradigm. Active message request handlers must be attached to each instance of the SPMD program by calling a collective operation `gasnet_attach()`. Active message request handlers categories include short, medium, and long, depending on the size or argument lists. The extended API is meant primarily as a low-level compilation target and can be implemented either with only the core API or by leveraging higher-level primitives of the network interface cards. The extended API includes *Put*, *Get*, and remote `memset()` operations. Data transfers are non-blocking, and the synchronization barrier is split phase.

Conclusion & Problems for Clustered Manycores

However useful, classic HPC communication layers cannot be effectively applied to manycore processors with local memories because of three differences:

- The memory capacity locally available to a core is about several gigabytes of memory on HPC systems, while it is tens or hundreds of kilobytes on manycore processors.
- HPC communication libraries assume a symmetric memory hierarchy, where the total memory is the union of the compute nodes memories. Manycore processors not only have (on-chip) local memories but also one or more external Double Data Rate (DDR) memory systems.
- A network-on-chip interface is much less capable than a macro network interface, but it has significantly lower latencies.

4.1.3 "Asynchronous Copy" Primitives of OpenCL

OpenCL is an acceleration programming model available on the targeted clustered manycore of this thesis, as already seen in Section 3.2.2. OpenCL structures a platform into a Host connected to Compute Devices. Each Compute Device has a main memory, which is shared by Compute Units. Each Compute Unit has a local memory, a cache for the main memory, and Processing Elements (PEs) that share the local and the main memories. Each Processing Element (PE) has registers and private memory. Computations are dispatched from the Host to the Compute Units as Work Groups. A Work Group is composed of Work Items, which are instances of a computation kernel written in the OpenCL-C dialect. This dialect includes vector data-types and requires to tag memory objects with their address space: `__global` (main memory), `__local`, `__private`, and `__constant`.

Performance Problems for Clustered Manycores

For clustered manycore processors, the main shortcoming of OpenCL is the inability to support efficient communication between the local memories and synchronization between the Compute Units. Communications between the local memories avoid additional memory copies in the main memory and provide important speedups. This capability is essential for efficient implementations of image processing, Convolutional Neural Network (CNN) inference and other algorithms where tiling is applicable.

Performance Optimizations for Clustered Manycores

OpenCL was originally designed for the GPGPU manycore architecture, where context switching at the cycle level is exploited to cover memory access stalls with useful computations. But Digital Signal Processors (DSPs) and static scheduling Central Processing Unit (CPU)-based manycore architectures do not implement such context switching at the cycle level. To target these architectures, OpenCL defines asynchronous prefetch or copy operations between the main memory and the local memory. The programmers can manually use them to build processing pipelines to overlap communications with computations. More specifically, OpenCL defines the `async_work_group_copy` and `async_work_group_strided_copy` operations. The asynchronous copy operations of OpenCL have proved highly useful in order to exploit Direct Memory Access (DMA) engines available on Field-Programmable Gate Arrays (FPGAs), DSPs or clustered manycore processors like MPPA®.

The main limitation of these operations is that data must be read/written contiguously from/into the local memory (dense mode), an assumption that turns out to be overly restrictive. For instance, an image processing tiling decomposition may need to copy a 2D sub-image of 16×16 pixels to a larger local buffer, allocated at 18×18 pixels to deal with halo pixels. In this case, one must explicitly manage a local stride of two pixels between each data block, since the local buffer is sparse and data should not be written contiguously to it. This restriction of OpenCL asynchronous copy operations is even more apparent when local buffers are declared as true multi-dimensional arrays, as supported by the C99 and OpenCL-C standards. The use of multi-dimensional arrays particularly eases 2D/3D stencil programming.

As the OpenCL standard allows vendor extensions for efficient usage of the target hardware, an asynchronous 2D copy has been implemented on the OpenCL runtime of STHORM P2012 [LPF13]. Other companies such as Xilinx®, Altera®, Adapteva®, AMD®, and Intel® also provide similar OpenCL extensions on their processors.

Conclusion for Optimizing OpenCL Applications

The targeted manycore architecture of this thesis already supports OpenCL, but not the asynchronous copies which are the `async_work_group_copy` and `async_work_group_strided_copy` operations, and other extensions of them. It is essential to make them usable for performance optimization.

4.2 Two-Sided Communications

send/receive and *read/write* operations characterize the two-sided communications. Two-sided communications are different from one-sided operations as either part of the communication link has to initiate explicitly a send and the other a receive operation. Because of that, two-sided implementations are sometimes too strongly coupled, leading extra waits on processes. However, it is still possible to make them efficient when performing asynchronous data transfers or when multiple threads initiate transfers, thanks to the release of the CPU internally if available (depends on the API providers). Figure 4.1 shows the semantic of two-sided communications.

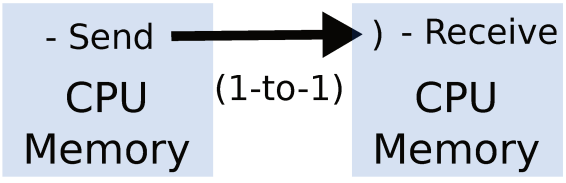


Figure 4.1 – *Two-Sided Communication Examples. The sending transfer initiated by the left CPU must strictly match the received command initiated by the right CPU.*

4.2.1 Rendezvous

Rendezvous protocol enables computing resources to find each other over a communication link. Rendezvous allows to synchronize resources, and data are sent and received only when either part of the communication link are ready. Thus rendezvous is a blocking communication protocol that can become an issue for performance due to the wait. Indeed

if the latency is high and the communication granularity is fine, the bandwidth of the communication link might be misused. Hence, Rendezvous is usually used to exchange large messages, so the implied latency of the transaction is far shorter than the time to transfer the data.

4.2.2 Synchronous-Asynchronous Send/Receive Protocol

The send/receive protocol covers a large panel of communications and allows either blocking rendezvous or asynchronous sends that are received on the other side by the corresponding receives. The local completion of the send in a blocking call is given when the function returns, or the transaction on non-blocking call completes when the associated wait returns. The two-sided send/receive operations are nevertheless the main communication primitives proposed by MPI [LJW+04].

4.2.3 Problem of Strict Matching

The send/receive protocol is sometimes complex to use when trying to optimize communication performance because of the strict matching requirement as shown in Figure 4.1. Strict matching means that send/receive operations need to be synchronized all the time before initiating communications. The matching must be done in the send/receive protocol to allow a send transaction to be linked with a receive transaction [IHIY14]. Therefore, such synchronizations sometimes increase the number of transactions at user level making applications complex to implement. Adding synchronizations imply CPU stalls that can severely limit the performance of applications. Asynchronous (non-blocking) data transfers are then available for optimized executions. The programmer can interleave communications and synchronizations by hand using these asynchronous send/receive operations. This task is not trivial and error-prone. Automating these operations in a compiler would help the programmer to optimize the execution of its applications.

4.3 One-Sided Communications

One-sided communications are defined by the fact that the data transfer can be executed without any application software involvement of the remote process memory. The initiator of the data transfer is a master of the distant memory that must be read or written. It is either a *read* or a *write* operation. Figure 4.2 shows the well-known protocols *Load-/Store* for fined-grained data communications and the *Put/Get* for coarse-grained data communications, both described in Section 4.3.1 and Section 4.3.2 respectively.

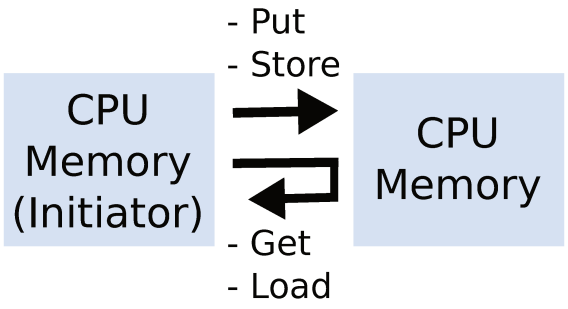


Figure 4.2 – *One-Sided Communication Examples*

4.3.1 Load/Store

A *Store* takes data from the core *register file* and writes it at the requested memory address. A *Load* requests data at a memory address and writes it in the *register file*. *Load/Store* is the simplest method to transfer data.

Any memory access that has the correct MMU mapping regarding virtual and physical addresses allow the processor to access a memory segment through the cache hierarchy. Manycore architectures either support transparent access to the main memory through hardware caches or via MMU-based distributed shared memory system like TreadMarks [KCDZ94].

However, on large-scale parallel systems, *Load/Store* through multiple levels of cache hierarchy can be a severe performance bottleneck when there is data sharing between numerous Non-Uniform Memory Access (NUMA) nodes. It does not scale easily because of the cache coherence traffic if supported. In particular, the implementation of reductions and inter-core/node communications are complex to perform efficiently.

4.3.2 Put/Get Remote Direct Memory Accesses (RDMA)

RDMA is the backbone of distributed memory systems when "long-range" data transfers are required. "Long-range" data transfers are usually large memory accesses (greater equal than 64 bytes), over a network between two CPUs, and DMAs perform the effective data transfer. A DMA is a hardware unit which can read and write in the memory by itself, once it has been programmed by software.

RDMA is a one-sided communication protocol inspired by [NTKP06], [VCHP07] and [KP03]. Any communication initiator registered to a memory segment is a master on this memory. A RDMA transfer can be initiated over a target memory segment using either the *Put* or *Get* primitives. However, RDMA operations favor high-throughput over low-latency.

However, enabling RDMA operations still requires the management of the memory consistency and the synchronizations. Examples in real-life applications and key features of RDMA are shown as well as principles when using it in software implementations.

RDMA Features in Modern Systems:

- OS-bypass allows direct interaction between the application and a virtualized instance of the network hardware, without involving the operating system.
- Zero-copy enables a system to place transferred data directly to its final memory location, based on information included in the RDMA operations. Zero-copy is also possible for Send/Receive protocols.
- One-sided operations allow communication to complete without the involvement of the application thread on the remote side.
- Asynchronous operations are used to decouple the initiation of a communication from its progress and subsequent completion, to allow communication to be overlapped with computation.
- A relaxed memory consistency model is typically applied [IHIY14], and it is explained in details in Section 4.4.2. A Relaxed memory consistency model allows operations to execute asynchronously with out-of-order global completion. For comparison, *Load-/Store* is also a one-sided operation which makes the RDMA protocol usually more natural to manipulate than classical *Send/Receive* where an overhead exists because

of the strict matching of the operations *Send* and *Receive* (need to synchronize all the time).

RDMA in Most HPC Systems

These communication technologies mostly apply at the backplane and system levels in data centers and supercomputers:

- Inside a compute node, between cores and other bus masters, HT (HyperTransport), QuickPath Interconnect (QPI) and PCIe support load/store as well as DMA (Direct Memory Access) operations.
- Between compute nodes across a backplane or a chassis, Serial Rapid Input-Output (sRIO) and Data Center Bridging (DCB). Ethernet variants such as RoCE (RDMA over Converged Ethernet) mostly support RDMA operations.
- At the system level, between compute racks and the storage boxes, Infiniband or Ethernet also support RDMA operations.
- Between systems, IP networks support the BSD sockets and client/server operations.

4.3.3 Remote Atomic Operations

A remote atomic operation consists of an initiator sending a message with an operation using different operands to a remote or external computing resource. Then, this remote computing resource executes the atomic operation possibly under certain conditions and forwards the completion to the initiator. Atomic operations are explained in Section 4.5.4.

Remote atomic operations have proved to be efficient for inter-node synchronization[MRSD16, BPG01] and more generally for parallel programming. For instance, they can provide classical atomic remote instructions such as *fetch-and-add*, *compare-and-swap* and *load-and-clear*. Such features are fundamental in distributed memory systems. Indeed most applications require an atomic update of variables at some point during the execution on massively parallel systems. It is also the case on shared memory architectures.

4.4 Memory Consistency & Coherence

Memory consistency and coherence are two different but fundamental concepts in parallel computing. This section explains how memory systems behave and how to use them properly by keeping in mind what happens in shared or distributed memories. For more information regarding consistency and coherence of memory systems, refer to [SHW11].

4.4.1 Definitions

Consistency and coherence of memory systems are defined [SHW11] as follows:

Definition 4.4.1.1 (Consistency)
Consistency models define correct shared memory behavior concerning loads and stores (memory reads and writes), without reference to caches or coherence. The consistency model defines a set of rules and a contract that must be respected between the software thread and the memory system. The contract is a set of information events of the memory system guaranteeing the initiator the state of the memory regarding initiated reads and writes operations to the memory system.

Generally, a consistency model [SHW11] defines rules to be followed by the programmers to guarantee that shared resources of a concurrent system will obey the contract (consistent).

Consistent means that the shared resource must be accessed atomically to prevent the systems from races and corruptions. A race is observed when a variable in memory is accessed concurrently without using a protection mechanism. Therefore, consistency implies two important pieces of information regarding the access of a shared resource. First, the acknowledgment of the accessed shared resources and secondly a synchronization effect with other initiators that can access this shared resource.

Consistency models are necessary for shared or distributed memory systems accessed by different initiators, such as processing elements or external mapped peripherals. As such, the consistency of shared resources is also applied to the registers of hardware peripherals, making their accesses not a trivial task in concurrent systems.

Definition 4.4.1.2 (Coherence)
A raw definition of coherence is that any effective modification of a memory address by an initiator must be seen by other initiators reading this memory address if no other write occurred at this memory address. From Hennessy and Patterson [HP11], it consists of three invariants: (1) a load to memory location A by a core obtains the value of the previous store to A by that core, unless another core has stored to A in between, (2) a load to A obtains the value of a store S to A by another core if S and the load "are sufficiently separated in time" and if no other store occurred between S and the load, and (3) stores to the same memory location are serialized.

Coherence is a generic word to describe if a system is coherent or not; however, it is usually applied to memory systems. Coherence is applied at several data cache levels until the Last Level Cache (LLC), but it is also applied to instruction caches for fetching the proper instructions in case of relocation [VWM04] for instance. Moreover, coherence is also found when a DMA modifies the memory, when there is Translation Lookaside Buffer (TLB) replacement in multi-threaded programs, or when the Linux kernel checks if the hardware MMU is coherent with the software MMU. As previously said, sometimes the software thread must deal explicitly with coherence and not only in multi-threaded programs. For instance, when managing DMA writes and core reads, if the hardware does not deal with implicit cache invalidation of the modified addresses range by the DMA, the software will be in charge of doing the invalidation, only when the address range is going to be updated in the memory.

4.4.2 Memory Consistency Models

The memory consistency model is related to the memory system architecture and the initiator(s) of the memory transaction. The simplest memory consistency model is the sequential consistency.

Definition 4.4.2.1 (Sequential Consistency)
The sequential consistency guarantees that all Load and Store operations are executed in total order regarding the program order of a software thread. Sequential consistency allows that a single software thread (and only a single software thread) can never corrupt data by itself when the memory system reorders memory accesses for performance.

Relaxed Memory Consistency for Performance

The term *relaxed* means that memory accesses can be out-of-order even if memory reads and writes happen sequentially in the calling thread. A relaxed memory consistency model provides better performances, but it is more complicated to use regarding the software. The additional complexity of relaxed memory consistency models is associated with the software in charge of explicitly waiting for the completion of the memory system.

Relaxation is the foundation of the performance of memory systems in parallel computing. One-sided operations like *Load/Store*, *Put Get* and remote atomic transactions can be reordered for performance. Reordering allows simpler or smarter implementation [SHW11] of the arbitration of memory transactions for instance. In high-performance memory systems, memory transactions are scheduled in parallel at different addresses, and depending on conflicts some resources can be slower than others regarding the initiator of the one-sided operations. But the shared resources always sustain the highest possible throughput. Depending on the implied targets and initiators, outstanding transactions usually complete out-of-order, providing better performance as Read-After-Write (RAW) dependencies are satisfied earlier, so that the computation can start as soon as possible. The RAW dependency defines the data dependencies of a computation. Therefore, one-sided operations benefit from a relaxed memory consistency model whereas two-sided operations do not because of the strict-matching. However, two-sided might benefit from reordering but at a lower level, like onto RDMA channels of an Infiniband network as in [LJW+04].

Memory Consistency Models in Real Life

The most used and implemented memory consistency model is called the Total Store Order (TSO). TSO is implemented on x86 Intel, AMD and Sparc architectures which implement a store First-In-First-Out queue (FIFO) to write data to caches. This store FIFO, also called *write-buffer*, also plays the role of coalescing data writes. TSO provides a relaxed *write* → *read* ordering. This FIFO is known as a *write-buffer*, whose purpose is to coalesce memory accesses. Other well-known memory consistency models exist such as the Partial Store Order (PSO) which relaxes *write* → *read* and *write* → *write*, and the Relaxed Memory Order (RMO) where all combinations of *read* and *write* are relaxed. The relaxed memory consistency model (RMO) is used on the MPPA® processor.

4.4.3 Memory Fences

Memory fences are fundamental in parallel software as they let the memory system be "consistent" at a certain point in the execution thread. Memory fences are part of the Instruction Set Architecture (ISA) of a core. The ISA defines all the supported instructions of a core. Memory fences provide the calling thread with the information that read and/or write operations are all completed. Also, fences can be commands of peripherals like DMAs that require memory consistency points at the end of transactions. Several types of fences exist:

- Write fence: waits for the completion of all outstanding writes initiated by the calling thread to the memory system. From that point, no memory system reordering is possible regarding previously initiated write transactions. On x86 it is mapped onto the `sfence` instruction.
- Read fence: guarantees for the calling thread that all previous reads to the memory have completed and are available in the core. The Read-After-Write (RAW) depen-

dency is satisfied on completion, the data is then available at the register level. On x86 it is mapped onto the `lfence` instruction.

- Full memory barrier: gives the completion of all outstanding read and write to the memory system. All previous read and write transactions are completed including outstanding atomic instructions before any other memory transactions can be initiated. On x86 it is mapped onto the `mfence` instruction.
- Pipeline barrier: implements a full memory barrier and waits for all on-going instructions in the core pipeline to be complete.

In the standard GNU Compiler Collection (GCC) atomic, the full memory barrier is available using the builtin `__sync_synchronize()` [Doc07] that will generate the `mfence` instruction onto x86 architecture. At high levels of multi-thread programming, memory fences are done by synchronization primitives provided by Multiprocessor System-on-Chip (MPSoC) vendors in runtime libraries which are encapsulated in standard libraries, e.g. the famous GNU C Library [SLwRMSD18].

4.5 Managing Current Memory Accesses for the Kalray VLIW Core

Managing the memory consistency of concurrent programs is not a trivial task when programming at low-level, that is without using high-level pthread-like synchronization primitives. In high-performance implementations, lock-free algorithms are designed to avoid the overhead of software serialization. Such implementations are difficult to write and validate because of transitional states and concurrency. In this section, the memory consistency management of the k1 VLIW is explained in details, making Chapter 5 and 6 easier to understand.

4.5.1 Cache of the k1 VLIW Core

The data bus size of the k1 VLIW core is 64-bit. Each core can exchange 8 bytes per cycle half-duplex. The k1 VLIW core implements a single level (L1) non-coherent 2-way associative data cache of 8 KB, with a data cache line size of 64 bytes. The non-coherent data cache means that modified data in upper memory levels will not be seen by this non-coherent data cache without software interventions. The instruction cache is also a single level cache and it has the same properties as the data cache.

The data cache of the k1 VLIW core implements a true Least Recently Used (LRU) policy for the eviction of cache lines. The data cache implements a *write-through* policy, using a *write-buffer* of 8 slots of 64-bit (8 bytes), whose purpose is to coalesce writes in the second level memory, which is, on MPPA, the local memory of the Compute Cluster (CC). Indeed the k1 VLIW processor is fitted with a Load-Store Unit able to deal with cached accesses of size 1 (byte), 2 (half), 4 (word) or 8 (double) bytes. As such, when small stores (less than 8 bytes) are initiated, the *write-buffer* will absorb all of them if possible, to avoid consuming local memory bandwidth for memory transactions that are less than the size of the data bus, which is 64-bit. Moreover, the data cache is not *Write-Allocate*, meaning that if a cached store does not hit in the data cache, the data cache does not miss. The write only goes in the *write-buffer* before being committed to the upper-level memory (local memory). That is why the k1 core cache is usually called to implement a *write-through write-around* policy. The only way to bring a data cache line in the data cache is to emit

a cached load at a cache-able memory address. On a cache hit the read-after-write data dependency is 2 cycles, whereas on cache miss it is 11 cycles critical word first on aligned memory access. The maximum number of outstanding miss (on-fly refill) is 1 with no hit under miss support.

4.5.2 Streaming Memory Accesses

The k1 processor implements streaming memory accesses, providing the ability to bypass the L1 data cache of the CPU for both the cache and *write-buffer*. Streaming memory accesses aim to gain performance regarding the following points:

- Prefetch of data. Streaming loads have a higher latency of 10 cycles for the read-after-write dependency onto which the load is supposed to return. As such the read of the register has to be placed at least 10 cycles after the initiation of the load to avoid k1 core stalls.

- Avoid trashing the data cache on sparse memory accesses. On sparse memory accesses that are not at the geometry of the data cache, the L1 is trashed and useless as no data locality is found. As such the streaming memory access makes it possible to access and prefetch the data without involving the L1 data cache.

Streaming memory accesses are used in data-intensive and compute intensive application to eliminate core stalls on data cache misses. For instance, streaming accesses are often used in image processing, signal processing, linear algebra and also the low-level software runtime to reach high-performance.

Also, the k1 core implements two types of streaming memory access: blocking and non-blocking. The problem with non-blocking is that when debugging an application, the core trap (exception) on memory access is not precise. It means that the architectural information inside the core will not be given (bundle, memory access type, the address). Thus, for debugging, we usually set the streaming in blocking mode and after, once the application is functional, the streaming accesses are then enabled for performance. The blocking mode provides only 1 outstanding load, whereas, in non-blocking mode, a streaming load request FIFO of depth 10 (10 outstanding loads) makes it possible to absorb the memory latency of the local memory from the k1 core.

The k1 VLIW core provides trapping and non-trapping streaming memory accesses of size 1, 2, 4 and 8 bytes. Non-trapping loads, also known as speculative loads, allow the developer or compiler to optimize memory accesses statically (compile time) by initiating loads onto unchecked pointers or for managing unrolled loop remainders without any checks. The semantic of non-trapping loads consists of loading to an address, and if the address has no mapping in the MMU, the load silently fails (no traps are triggered). Then the core places zero into the targeted register(s) where the loaded data was supposed to return. The user code or generated code is then in charge of controlling the side effects.

4.5.3 Managing the Coherency & Consistency of the k1 VLIW Core

The k1 core implements a relaxed memory consistency model for performance thanks to the *write-buffer*, the multi-banked local memory, and the streaming memory accesses. The ordering between reads and writes is ensured for cached accesses alone (only), streaming accesses alone (only) but **not** for both cached and streaming memory accesses.

The k1 core is fitted with the following cache management instructions.

- `wpurge`. Request the *write-buffer* to commit in the local memory all outstanding writes. The committing of dirty bytes in the local memory goes into the memory system and will be updated in the memory hierarchy in the next clock cycle (pipelined).

- `fence`. This memory fence operation waits for the completion of all outstanding writes to the memory for the *write-buffer* of the data cache or the streaming writes. When the `fence` instruction returns, the core is guaranteed that the memory hierarchy is updated and consistent for other memory masters on the targeted memory. The combination of the `wpurge` instruction followed by the *fence* instruction is equivalent to a full runtime write memory barrier on the k1 VLIW core.

- `[d,l]inval`. The data cache invalidation instructions allow the k1 VLIW core to see writes of other masters in the memory system. Two instructions are provided, the `dinval` instruction invalidates the entire data cache, meaning that the entire data cache will be cold after the operation. Thus, next loads on the *stack*, the *heap*, the *.data* section or even the *.bss* section will miss. For finer-grained coherent reads, the `linval` instruction can be used, which invalidates a single data cache line address.

- `iinval`. The `iinval` instruction operates onto the instruction cache. On statically linked code, the coherency of the instruction cache does not need to be managed. However, invalidation of the instruction cache needs to be done when the core relocates code as the memory containing the code is modified. The `iinval` instruction must be followed by a pipeline barrier to make sure that, when the core resumes the execution, it will fetch the proper instructions from the memory system.

4.5.4 Atomic Instructions

An atomic operation is defined by atomically updating a data in memory using an *atomic read-modify-write* operation. Atomics are usually simple operations like *fetch-and-add*, *test-and-set* or *compare-and-swap*. *Compare-and-swap* makes it possible to perform any *fetch-and-OP* operations. Some atomic operations are available in the hardware ISA, and those, not supported in hardware, have to be implemented in software. For instance, GCC implements the *libatomic* to implement software atomic operations when not supported in hardware. Obviously, the software implementation will have worse performance but the parallel software continues to work.

Efficient implementations of lock-free systems exist such as [LMS04] and [Mic02]. High-performance parallel software cannot avoid lock-free implementations, but depending on the application, it can be very complicated to design, verify and debug. The k1 VLIW core implements both retry-free and lock-free atomics. In this section, we consider only uncached atomics on both Compute Cluster (CC) and Input/Output Subsystem (IO) side. Indeed the Input/Output Subsystem (IO) multi-core provides cached atomics that are performed in the shared L1 data cache (Section 2.3.2) and explained in patent [DDRV17].

Several atomics exist and they are explained below.

Blocking Atomic - Software

The thread takes a lock, performs a full memory barrier, then the thread performs the operation in memory, performs a full memory barrier and releases the lock. Blocking atomics are widely used on simple multi-thread programs. When the lock is taken by a

thread, all other threads trying to take the lock stall until it is released. Only one thread can run inside the locked code section, also called critical region or protected region.

Lock-free Atomic - Partially Hardware

Forward progress is a guaranty for one initiator. Lock-free atomics are implemented using software loops as the atomic might fail when there is pressure at the memory address where the atomic is performed. On failure, the initiator must retry. An example of the k1 VLIW core is provided as follows.

- *Atomic-Compare-Word-and-Swap.* The use of this instruction is illustrated in the code Figure 4.3. The C code shows how to use the *Atomic-Compare-Word-and-Swap* instruction of the k1 VLIW core, called the *ACWSU* instruction. In this code, a value at an address is read (old value at Line 6), then the k1 VLIW core performs one or several operations on this value (new value at Line 8), and tries to write it (new value at Line 10) in the memory system. The initiated write to the memory system presents the old and new value, and the memory system compares if the current value is equal to the old value or not. If yes, the current value in the memory system is replaced by the new value in the memory system (Line 12), if not, the address in the memory system is left unchanged and the current value is brought to the k1 VLIW core, for further attempts (Line 13 comments). On the k1 Bostan VLIW core, the *Atomic-Compare-Word-and-Swap* operation operates on 32-bit operands, such as integers (int) or single-precision float numbers (float).

```
1. static __uncached volatile float acc = 0.0f; // uncached shared data
2. #pragma omp parallel // parallel region
3. {
4.       do {
5.               // volatile forces the compiler to load ``acc''
6.               float old_value = acc;
7.               // any operation
8.               float new_value = old_value + 1.0f;
9.               // atomic-compare-and-swap-uncached
10.              float current_value = ACWSU(&acc, old_value, new_value);
11.              if(current_value == new_value)
12.                      break; // success so we exit
13.              // retry as the value changed in memory
14.      } while (1);
15. }
```

Figure 4.3 – *Atomic-Compare-Word-and-Swap in Practice on the k1 VLIW Core.*

Retry-free Atomic - Fully Hardware

Also known as a wait-free atomic, forward progress is always guaranteed within a bounded amount of time for all initiator executing the atomic at a given address. Wait-free atomics have strong properties for the implementation of time-critical systems as the Worst-Case Execution Time (WCET) is bounded. Examples for the k1 VLIW core is provided as follow.

- *Atomic-Load-Double-and-Clear.* This instruction is an atomic read-modify-write instruction with high throughput. It consists in reading a value at an address in the

memory system, sending it to the k1 VLIW core, and resetting to zero the content of the address read in the memory. The *Atomic-Load-Double-and-Clear* operation is typically used to implement efficient locking mechanism in memory. The operation takes an address of where to read and clears to zero the data in the memory. The instruction is called the *ALDCU*.

- *Atomic-Fetch-Double-and-Add.* This instruction is an atomic *fetch-and-add* in the memory. The instruction reads the data in the memory performs a signed addition with a 32-bit operand, writes the new value in the memory system, and the old value in the memory system is placed in the destination registers (a pair of 32-bit registers as it operates on double words) of the fetch operation. The instruction is called the *AFDAU*.

Atomic instructions play an essential role in parallel computing as they allow multi-thread programs to synchronize cores, and to perform efficient reductions, and for lock-free resource sharing. High-performance parallel implementations try to use as much as possible retry-free atomics. However, they are not always usable, depending on performance needs, and they have a real cost in hardware.

On MPPA®, we use by default the *compare-and-swap* for the sake of simplicity. Indeed, the `ACWSU` instruction makes it possible to perform any *fetch-and-OP* in the memory system atomically, but at the cost of a software loop, as seen in the code in Figure 4.3. Atomics that are implemented with software loops can be time-consuming when there is a lot of contention on the memory bank where the atomic is being executed; even though, forward progress is always guaranteed for one initiator of the atomic. For code portability, the usage of standard builtin atomics like [Doc07] or C++ 11 atomics is recommended.

4.6 Conclusion

Data communications and synchronizations are essential on distributed memory architectures. The targeted clustered architecture is often known as a stream-based processor as all data movements are up to the software because of the lack of a global cache system supported in hardware. However, the fact that this processor implements local memories, makes it a serious competitor in low-power computing with deterministic timing-responses.

Therefore, we explained in this chapter the state-of-the-art of high-performance communication systems applied to diverse parallel programming models like OpenCL, PGAS, MPI and low-level ones such as the RoCE and other families of APIs. Our focus is put on the one-sided and two-sided communications. Details about the two-sided and one-sided protocols are highlighted such as rendezvous, *Load/Store*, RDMA *Put/Get* operations, and remote atomic operations.

The Kalray MPPA® processor implements in hardware a two-sided Network on Chip (NoC) for hardware simplicity. The two-sided protocol being difficult to use by programmers, we decided to design and implement a one-sided communication API over the MPPA® NoC. This implementation is explained in details in Chapter 5. The goal of the new one-sided communication API is to ease the software development and provides high-performance. The challenge in the development of the one-sided communication API is to adapt these well-known protocols to a local memory based machine like MPPA®efficiently.

In this chapter, we thus leveraged the understanding of the memory system of the MPPA® to make our contributions at both system and application levels as efficiently as possible. We explained prerequisites about memory coherency and memory consistency on parallel machines. We defined the memory consistency model of the MPPA® processor

and the x86 as it is the most used architectures. Low-level details have been given for the Kalray VLIW core about the management of the memory consistency and coherency. Cache management operations, atomic instructions, the implied performance latency, and usage semantics have also been explained. Such details are required to understand high-performance implementations in general and the following contributions of this thesis in particular.

Part II

Contributions

CHAPTER 5

Fundamental Mechanisms for Communications and Synchronizations in Distributed Computing

This chapter introduces fundamental mechanisms for communications and synchronizations applied to distributed computing on manycore processors. Manycore processors are composed of multiple Symmetric Multi-Processor system (SMP) machines also called clusters, which are a group of Processing Elements (PEs) sharing a coherence domain or a local memory. Inside an SMP machine, the communication and synchronization are handled by the *Load*, *Store*, *Atomics*, and *Fence* operations as explained in Section 4.3.1. At the level of multiple SMPs, where distributed computations and memories are found, the need for communicating and synchronizing is fundamental. Sections 4.2, 4.3.2, and 4.3.3 present mechanisms to communicate and synchronize programs distributed across clusters, such as the Remote Direct Memory Access (RDMA) *Put* and *Get* operations, the *Atomic* operations, the *Send/Receive* operations over channels and the *Fence* operations [NTKP06]. Modern communication technologies like Infiniband [Sha03] and recent evolutions of Peripheral Component Interconnect Express (PCIE) [Aja09] also implement these operations, as they provide the foundation mechanisms for distributed communications and synchronizations.

Prior to this thesis, the targeted manycore processor has PEs that already supports *Load*, *Store*, *Atomics*, and *Fence* instructions but only at the SMP level, inside a Compute Cluster (CC) as explained in Section 2.3.2. At the level of multiple SMPs (multi-CCs), the hardware only supports simple *Send/Receive* operations with massive software assistance to properly configure the Direct Memory Access (DMA) interface and the Network on Chip (NoC) on both sides of the two-sided communications, as explained in Section 2.3.3. In this chapter, we propose a comprehensive distributed software runtime and Application Programming Interface (API) that leverages the RDMA, *Atomic*, *Send/Receive* channels, and the *Fence* operations on top of a two-sided network of SMPs for a manycore architecture.

Two software solutions with production maturity were available at the beginning of this thesis for inter-clusters communications and synchronizations: the Multi-Purpose Processor Array (MPPA)® Inter-Process Communication (IPC) and the MPPA® NoC communication libraries. The IPC library [dDdML+13] is the oldest one and supports only *Send/Receive* communications. The IPC library does not split the control path and the data path; thus, several tens of thousands of machine cycles are required to initiate any data transfer, making it inefficient for medium-grained distributed computations. Also,

this library does not abstract the hardware DMA and NoC details, making it difficult to use for application software engineers. The MPPA® NoC communication library is even more complicated to use as it exposes the low-level details of the DMA interface and NoC. Data management and synchronizations are tough to handle for a non-expert of architectural details. However, the biggest issue with these libraries is that they only implement two-sided communications instead of one-sided communications. As explained in Section 4.2.3, in the MPPA® processor two-sided communications are difficult to use efficiently because of the strict matching issue implied by the two-sided protocol [IHIY14].

In addition to these solutions, there have been several failed attempts to port APIs like Message Passing Interface (MPI) or SHEM for this target manycore processor. In particular, MPI is the most successful standard in High-Performance Computing (HPC) systems and is designed for distributed memory machines. As seen in Chapter 4, this failure is mainly since these supercomputing APIs assume gigabytes of memory per SMP nodes. However, clustered manycores, like the CC of the MPPA®, integrates 2 megabytes of local memory.

We propose and describe in this chapter a new lightweight communication library called Asynchronous One-Sided (AOS). This library implements one-sided communication protocols for better performance and ease of use. This new AOS API also implements two-sided communications, with remote queues [BCL+95] for the *Send/Receive* operations. As the Kalray NoC is implicitly a two-sided communication NoC, the implementation of one-sided communications is a challenge.

This chapter describes the design, the implementation and low-level details of the distributed runtime and API for communications and synchronizations targeting a clustered manycore architecture. In Section 5.1, we present an overview of the challenges for implementing such communication protocols onto the NoC of a manycore processor. Section 5.2 presents the AOS library at the programmer level. We detail in Section 5.3 the design, the algorithms and the implementation of the AOS library. The resources allocation regarding the DMA NoC interface is explained in Section 5.4. Performance results and discussions are provided in Section 5.5. Latest contributions in the MPPA® AccessCore toolchain regarding the AOS communication engine are explained in Section 5.6.

5.1 Challenges

Asynchronous one-sided communications have proven to be efficient for HPC workloads by overlapping communication with computation and with fundamental ordering properties. However, enabling such a feature on a heterogeneous distributed local memory architecture like MPPA® is a challenge.

Firstly, the AOS runtime system must manage hardware resources (memories, DMA Rx Tags, DMA Tx packet-shapers and DMA micro-cores), which are explained in Section 5.4. This management of resources must be done for both local and remote resources in a massively parallel environment.

Secondly, several features have to be abstracted and provided to the application programmers such as Quality-of-Service (QoS) configuration, synchronization and bindings at the creation of communication segments for any protocol without the need for the programmer of being aware of the NoC topology for all on/off-chip memories. All abstracted features must be spread across the number of potential initiators, registered to a memory segment explained in Section 5.2.

Thirdly, as mentioned in [dDdML+13], the abstracted one-sided protocols should not be limited by the number of physical hardware resources. Indeed, all PEs can communicate

with all PEs, and there are not enough hardware resources for that. These hardware resources should be translated (virtualized) to different kinds of "software" components such as memory segments management, RDMA, remote queues and automatic flow control without reducing the performance of the hardware. Such an abstraction frees the application programmer from managing physical hardware resources, and software job First-In-First-Out queues (FIFOs) congestion control, which is often a complex issue and a source of error.

Fourthly, the software RDMA engines must provide the programmer with ordering properties for outstanding transactions and remote atomic operations, and maintain the memory coherence and consistency (memory ordering) at synchronization points.

Finally, as distributed software might suffer from the congestion of software job FIFOs, software flow-control is required to avoid data races. To do this, AOS implements a mechanism called "all-to-all client-to-server flow-control" for the remote atomic operations and RDMA to avoid data and request corruption when congestion occurs. Such constraints make our new one-sided communication software implementation very complicated to conceive, debug, and validate while reaching for the theoretical maximum hardware throughput.

Not part of this contribution but it is a significant feature that has been tackled during this Ph.D. is the observability issue. Observability is a big problem on highly parallel processors. Therefore, we developed at the very beginning of this thesis the File IO library. This library is a tool to speed up and ease the development as it makes the debugging and the observability of the running parallel application simpler. The File IO library aims to hide the complexity to bring in and out data from the local memories of the MPPA® Multiprocessor System-on-Chip (MPSoC). To keep it simple, in the Operating System (OS) system call handler, we implemented stubbed functions that perform a remote procedure calls of functions like *open, read, write, close* (system calls). The NoC and PCIE communications are handled automatically. For instance, all functions operating on files or file descriptors require the support of system calls. It has been designed and implemented at low-level for maximum performance; but, it is a synchronous runtime library, easier to use but at a performance cost.

5.2 Design of Distributed Protocols of Communications and Synchronizations for the Programmer

The AOS library provides two protocols that are the new one-sided operations and the two-sided operations. Two-sided operations were already supported by the MPPA® IPC API and the MPPA® NoC. An example of one-sided operations is shown in Figure 4.2, and an example of a two-sided operation is in Figure 4.1.

The AOS library presents the manycore platform as a collection of execution nodes, called CCs and Input/Output Subsystems (IOs). These nodes are composed of PEs and their directly addressable local shared memory. There is a specific for the IOs which can additionally access an external memory called the Double Data Rate (DDR) memory as presented in Section 2.3.

5.2.1 Memory Segments

In our contribution, two types of memory segments are defined and can be used by the programmer. A memory segment is a directly addressable buffer from the PEs of a node to the mapped memory (for instance the local memory or the DDR memory). The first one is

the *window* memory segment that supports one-sided operations such as RDMA *Put/Get* and *remote atomic* operations. Memory *window* can also be found in the MPI standard [HDT+15]. *Put/Get* operations are low-latency and high-throughput RDMA operations that are usually used to transfer large buffers between nodes. *Remote atomics* are used to synchronize nodes, for instance with atomic add and atomic clear remote operations. The second type is the *remote queue* memory segment that supports two-sided operations such as the *enqueue* and *dequeue* operations (1-to-1 or N-to-1). *Remote queues* are mostly used to enable fine-grained control or to implement the master/slave model as in an acceleration programming model.

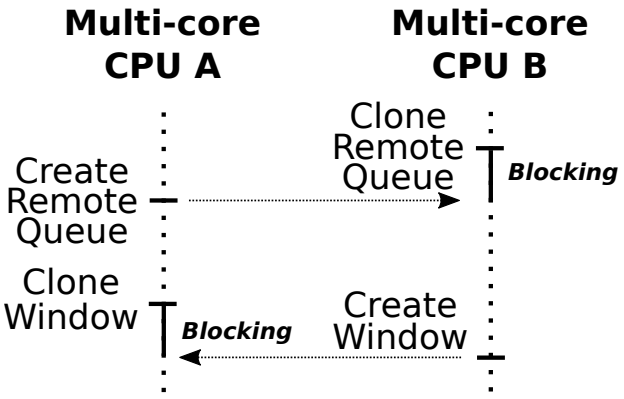


Figure 5.1 – *Memory Segment Usages with the Create and Clone Functions*

The creation operation of the memory segment exposes a part of the memory. The clone operation is then used by a PE to access the created memory segment. As seen in Figure 5.1, the memory segment has to be created and cloned between either part of the network to make the data communication possible. The NoC routes, addresses, DMA configurations, and offsets are automatically resolved at this time. The initialization of a communication link has to be carefully performed. The communication link is either a *window* or a *remote queue*. For this purpose, a node creates a memory segment, of a *window* type for instance, and this segment is associated with a unique 64-bit counter that is registered in a broker. The clone operation on this unique 64-bit identifier connects the initiator with the created memory segment. The clone operation implicitly provides synchronization with the creator. The communication can immediately start once the clone operation returns. Everything is managed by the internal software runtime enabling the AOS API. Figure 5.1 shows how the memory segment creation and cloning operations relate to each other.

5.2.2 One-sided

As seen in Section 4.3, one-sided communication provides the ability for a process to access a remote memory without any involvement in the communication at the programmer point-of-view. The one-sided communication functions that are provided to the programmer for the data path, are thread safe.

RDMA Operations

In our communication runtime, the RDMA operations are the most used to move data between the network of SMPs of the manycore architecture. RDMA transfers are made for high-throughput with low-latency. RDMA operations operate on segments of type *window*.

We define two primitives to perform remote read and remote write of data over the network respectively called *Get* and *Put* operations. Figure 5.2 shows the RDMA *Put* and *Get* principles between two nodes. The *Put* operation reads local data, and sends them to the network, in the cloned memory segment. The *Get* operation sends a remote read request using the cloned segment. When the remote read request is processed, the data are sent to the initiator (multi-core B) of the operation.

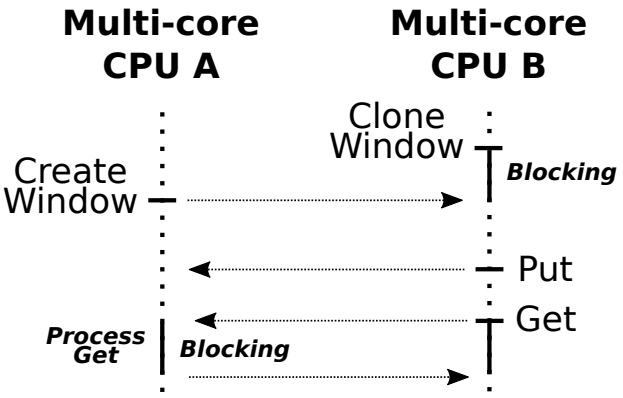


Figure 5.2 – *RDMA Put and Get Operation on Window Memory Segments*

From the programmer point-of-view, the *Put* and *Get* operations can be used either in a synchronous or asynchronous mode. When an RDMA operation is initiated, the synchronous mode gives the programmer the local completion when the calling operation returns. In the asynchronous mode, RDMA operations are executed in the background. The operation is associated with an event, given by the programmer, that can be tested or waited. Upon the completion of the event, the programmer is provided with the local completion, and not the remote completion.

The local completion of *Put* operations is given to the programmer upon completion of the operation, but this is not the remote completion where data are written. Remote completion or *Put* consistency is provided by the completion of an RDMA *fence* operation. The completion of the RDMA *fence* operation guarantees that all outstanding *Puts* are written and consistent in the targeted memory segment.

An important feature provided implicitly to the programmer is the flow-control of RDMA transactions. Flow-control mechanisms are complicated for the programmer to be implemented, which is why it is already provided by our contribution. Such mechanisms avoid the corruption of data when the distributed software implementation suffers from congestion.

Remote Atomic Operations

The remote atomic operation provides the programmer with fundamental and efficient mechanisms for distributed computing. The remote atomic operations are the foundation to perform: synchronizations, reductions, and collective operations between multiple SMPs of a network. An example of remote atomic operations can be seen in Figure 5.3. Once

the segment is created and cloned, *Put* operations and remote atomic operations can be performed. *Put* operations are performed asynchronously, and their remote completions are guaranteed by the *Fence* operation or other remote atomic operations.

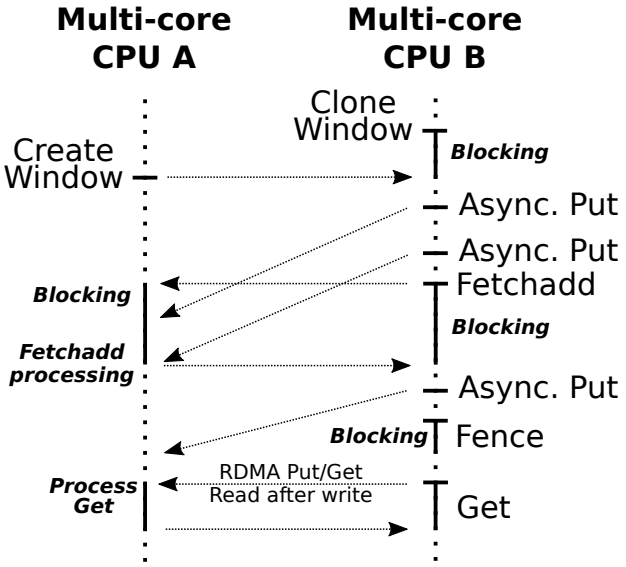


Figure 5.3 – *RDMA Put and Get Operations with Remote Atomics on Window Memory Segments*

We propose and provide to the programmer lightweight remote atomic operations such as *postadd*, *poke*, *fetchclear*, *fetchadd* and *peek*. Our panel of remote atomic operation is inspired by Infiniband [Sha03] and the supported atomic instructions of the targeted machine. All of these operations take as input a target segment of type *window* and an offset aligned 8 bytes (atomic alignment constraint) to which to remote atomic should be applied in the *segment*. The *peek* and *poke* operation are respectively uncached 8-byte read and write. The *postadd* operation atomically adds a signed value to a 64-bit counter in the remote memory without returning any result to the initiator. The *fetchadd* operation is the same as the *postadd*; but, it returns the read value of the *fetch-and-add* atomic instruction to the initiator. The *fetchclear* operation atomically clears the remote target value to zero, and the value is sent to the initiator.

The Ordering of the One-sided Operations

As seen in [IHIY14], one-sided operations make it possible for the hardware and the software runtime to relax the ordering of transaction. Indeed, one-sided operations can be reordered in the background because the initiator of the one-sided operation is a master of the targeted memory. The goal of such feature is to increase the performance of the overall distributed memory system by providing to the programmer a relaxed memory consistency model. In this section, we define a set of important rules of our new distributed communication runtime that must be known by the programmer:

- **Rule 1** The outstanding RDMA *Put* operations are strictly ordered from an initiator point of view about their local completion, but not for their remote completion (*fence*).
- **Rule 2** *Get* operations are ordered when reading from the same memory segment while reading from different memory segments is not ordered. Outstanding *Put* and

Get operations are not ordered on the same initiator for efficient parallel execution. Hence, for any reason, when Read-After-Write (RAW) dependency occurs (*Put* followed by a *Get* on the same memory segment), an RDMA *fence* completion must be performed before initiating the *Get* operation. The *fence* operation is provided by the one-sided engine and is part of the active message operations. In the memory consistency model, the completion of the *Fence* operation provides the remote completion of all outstanding *Puts* to the targeted memory segment.

- **Rule 3** Remote atomic operations are ordered if they target the same memory segment, else they are not. A powerful property with the outstanding RDMA transaction and the outstanding remote atomic operation is that they are ordered when targeting the same segment. An example of such an ordering is provided in Figure 5.3. It is obtained thanks to a point-to-point software "virtual channel" between each pair of segments. When an initiator (a PE) X posts several *puts* (RDMA transactions) and then posts a remote atomic operation to a memory segment, the posted remote atomic operation will be seen in this memory segment only after the remote completion of the previously initiated *puts* of the initiator X. Such ordering is essential for performance as the initiator can post high-throughput RDMA data transfers along with a posted synchronization mechanism. Indeed, everything can be done asynchronously from the initiator point of view. Therefore, this initiator can go back to computation immediately without losing any time. At the programmer level, this concept is called: the ordering between posted remote atomic operations and outstanding RDMA transactions.

5.2.3 Two-sided

As seen in Section 4.2, two-sided communications allow the programmer to send data to an opened communication channel and receive this data on the other side of this channel. Currently, the programmer is provided with low-latency remote queues that are explained below. The choice of remote queues has been conditioned by the need for low-latency and high-throughput (in terms of Input/Output Operation per Second (IOPS)) control queues at the user level.

Remote Queue Operations

For the programmer, the communication semantics of the remote queue is based on send and receive operations. The remote queues aim to be used for control messaging, usually small size messages. Small size messaging provides high throughput control concerning IOPS. These operations operate on segments of type *queue*.

Once the memory segment is initiated, the user can send messages to the queue using the *enqueue* operation and receive messages from the queue using *dequeue*. Figure 5.4 shows an example of remote queues for the programmer. Simple queue messaging uses 1-to-1 communication where only one initiator sends messages to a remote queue. However, the programmer is also provided with a remote queue mode where N-to-1 communications are possible. In N-to-1 mode, the atomicity of *enqueue* and *dequeue* operations is ensured for the user by our communication runtime. This N-to-1 capability is essential on massively parallel systems that require runtime orchestration of activities (e.g., master/slave parallel pattern).

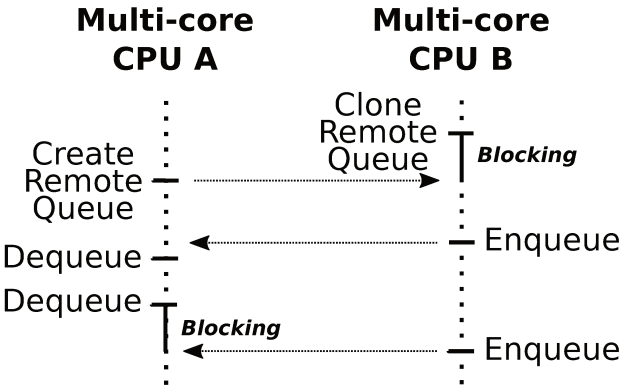


Figure 5.4 – *Enqueue and Dequeue Operation using Remote Queue Memory Segments*

Ordering

Two-sided operations feature the strict-matching property as seen in Section 4.2. Therefore, no reordering is possible between successive sends of data. For the programmer, the order of the transaction completion is the same as the order of the transaction initiation. In asynchronous mode, the order is strict for both *enqueue* and *dequeue* operations.

5.2.4 Restructuring: Data Layout

The contribution also features on-the-fly and zero-copy data restructuring functions for the programmer, using diverse modes of RDMA *Put* and *Get* operations. Such mechanisms are motivated by applications like computer vision, deep learning, signal processing, linear algebra, and numerical simulation, that require data restructuring to make their parallelization more efficient for clustered manycore architectures. Thus, the application programmer can change the layout of the data while performing the RDMA transfer. Changing the layout of the data is usually performed when copying data from the external memory to the shared local memory in the CCs. We looked at [CEL+03] to provide the programmer with a panel of transfer geometries, they use continuous, strided and repeated vector transfers for instance. We added a few more patterns, typically the *spaced-remote-sparse-local* which is an arbitrary *remote_stride* and *local_stride* offsets transfer, and its induced implementation for generalized 2D and 3D data blocks.

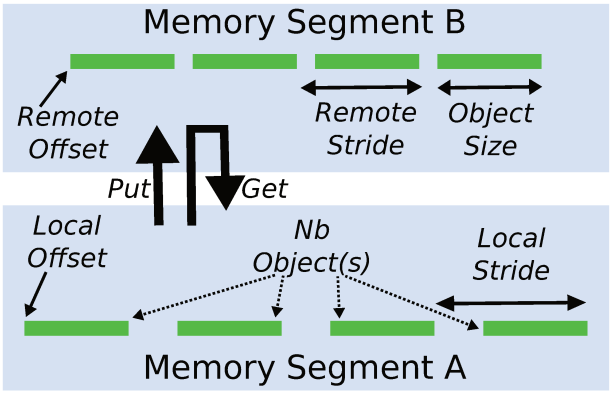


Figure 5.5 – *RDMA Put/Get Data Transfer Restructuring Pattern*

Data restructuring is generally used when using tiling techniques with or without overlap, halo region forwarding, transposition patterns, 2D, and 3D block transfers. 2D and 3D copies are special cases of strided copies, with stride-offsets on both *src* and *dst* buffers. These offsets can be different from each other, as the local buffer is often smaller and accommodates a sub-partition of the remote buffer. We call this capability the "on-the-fly data restructuring". Figure 5.5 shows the available RDMA *Put/Get* data restructuring opportunities without a copy. Such patterns are useful and efficient for image processing, Convolutional Neural Network (CNN) and many other stencil-based applications, where tiling is applicable.

5.3 Runtime Implementation of the Distributed Communications and Synchronizations

This section explains how the distributed communications and synchronizations library, called AOS, has been conceived and implemented. The implementation of the memory segment management is explained in Section 5.3.1. The RDMA implementation over the Kalray NoC and the implementation of the remote atomic operations are presented in Section 5.3.2. The event completion engine, used to get the completion of any asynchronous operations of the AOS API, is explained in Section 5.3.3. Section 5.3.4 provides the implementation of an efficient job arbiter to process the remote atomic operations and the RDMA operations. Finally, the remote queues, used for fine-grained two-sided communications, are described in Section 5.3.5.

5.3.1 Memory Segments

The creation and clone operations of the memory segments are used to initialize at user level a communication link. As already seen, the communication link can be either a remote queue or a memory window.

The segment creation and clone operations are handled thanks to a centralized broker that references all living memory segments for all CCs and IOs. Therefore, the creation and clone procedures ought to be performed by the programmer only at initialization time of the distributed computing system for performance, but can be used whenever the programmer needs them.

This centralized broker has been implemented using low-level message passing over the NoC of the manycore. A multi-core Central Processing Unit (CPU) provides an identifier referencing the created segment in this broker. Other multi-core CPUs can connect to the created segments using the related identifier at the *clone* operation.

5.3.2 One-sided: Asynchronous Remote Atomic Operation & RDMA Put & Get Algorithm

This section explains the implementation of the remote atomic and the RDMA operation. The implementation is inspired by a well-known mechanism used in distributed computing, called "active messages." An active message is a low overhead message that acts as a call, with or without returning a value and using a message payload as arguments. Paper [MRSD16] also uses similar mechanisms to implementation inter-core notifications.

The one-sided user API functions take as arguments: a local virtual address that is read or written, a remote target segment, transaction parameters (e.g., operation type, size, stride, geometries) and an event for the completion of the initiated transaction. The

RDMA functions are *Put* and *Get* operations, and remote atomic operations *postadd*, *poke*, *fetchclear*, *fetchadd* and *peek*. A *fence* operation is provided for the remote completion of outstanding RDMA transactions of a targeted memory segment.

Algorithm 2 Active Message Initiator Algorithm (thread safe algorithm)

```
1: Input: segment, operation, parameters, blocking
2: Output: event
3: if not_valid(segment) or not_valid_op(operation, parameters) then
4:    Return failure
5: end if
6: node_id = load_target_node_id(segment) /* Remote cluster ID */
7: amsg = Prepare active message transaction /* RDMA, Remote atomic or Fence */
8: Write memory barrier /* Commits outstanding user stores in local shared memory */
9: slot = atomic fetch add array_slots[node_id] /* Shared array in the cluster */
10: /* Flow-control: verify the number of outstanding transactions to avoid corruption */
11: while (slot + 1) >= (array_done_slots[node_id] + FIFO_SIZE) do
12:    Idle PE /* OS yield possible */
13: end while
14: /* Check local or remote transaction */
15: if is_remote_transaction(segment, amsg) then
16:    route = compute_route(segment) /* Compute network route */
17:    Tx_configure(route) /* Tx hardware resource is protected by a lock */
18:    Tx_send_message_notify(amsg) /* Remote memory in another cluster */
19: else
20:    job_fifo[(slot + 1) % FIFO_SIZE] = amsg /* Write job in local shared memory */
21:    Write memory barrier
22:    Broadcast notify to all PEs /* Local shared memory of this cluster */
23: end if
24: event = prepare_event(segment, slot)
25: if is_blocking(blocking) then
26:    Call Algorithm 3 (event) /* Blocking mode, wait for the event to occur */
27: end if
28: Return success
```

Algorithm 2 is used to send active message requests to the job arbiters, presented in Algorithm 4. The Algorithm 2 is executed by the PEs inside each CC, in the AOS library called by the programmer. We assumed that the input segment has already been created or cloned. The provided Algorithm is thread safe, and it can be run on any PE of a CC of the used manycore processor.

The *remote atomic*, the RDMA *Put* and *Get* are usually the critical path of data-intensive applications. All one-sided data transfers are managed by the Algorithm 2. Therefore, it needs to be efficient, thread-safe and programmer-friendly (e.g., management of maximum outstanding jobs with flow-control). The RDMA and *remote atomic* transactions operate on a *window* memory segment. Initiating a one-sided operation consists in parsing the targeted segment parameters (Line 6 and 16 in Algorithm 2) such as the requested segment protocol(s) (RDMA or remote atomics in our case), the NoC route, the destination DMA Rx Tag and checking if the read or write transaction is not out-of-bounds of the targeted remote *window*. Then the transaction is prepared by the initiator PE in a CC, and it takes a slot (Line 9) on the targeted segment atomically. An N-to-N flow-

control mechanism has been implemented for remote memory transactions "inter-node" and local memory transactions "intra-node" (array_done_slot Line 11 in Algorithm 2) which provides a back pressure mechanism when the hardware and low-level software is under congestion. Then, the PE sends the request either to the NoC interface or writes the request in the shared local memory. The completion ticket of the initiated transaction is computed and set to an event (Line 24) which can be waited for later, using Algorithm 3. An event is an opaque object that contains the necessary information to provide the programmer with the completion of the related initiated transaction. Indeed, the programmer can give a reference to an event to the algorithm, with a boolean flag "blocking" set to true. The algorithm fills it with the necessary information of the operation and returns immediately (the operation is outstanding). If this boolean flag is set to false, the algorithm will wait until the completion of the operation before returning.

In our implementation, RDMA operations are immediately processed with our software algorithms when the hardware resources are available. Remote atomic operations are processed when all previous RDMA *Puts* are completed. These operations have a posted variant (without a returned value), which can be very effective for the implementation of reductions, and synchronizations onto distributed manycore architectures.

5.3.3 Event Completion

The completion of an event is associated with an operation, previously initiated by a PE with the programmer. For instance, the operation can be an RDMA or a remote atomic transaction. The completion of AOS events is managed by Algorithm 3.

Algorithm 3 is designed for managing the event completion with fast execution time. An event is a condition that has only two states (*true* or *false*) for the programmer. This event contains an address (Line 3 in Algorithm 3) to monitor and compare it with a value (Line 2) using simple condition (Line 6) (e.g., *equal*, *greater*, *less*). Depending on the event type of the associated transaction, the event can complete by getting hardware pending events (Line 11 in Algorithm 3) and accumulate them into the content of address (Line 13). This sequence is required to prevent DMA Rx Tag End-of-Transfer (Eot) counter from saturation. However, this sequence has to be atomic; thus, we use atomic uncached instructions, and we notify all other PEs of the CC when the content of address is updated and visible in the memory hierarchy. This broadcast notifies operation (Line 16 in Algorithm 3) is done using a low-latency *control NoC Rx mailbox* in barrier mode which leads only to a single posted *store* in the peripheral space for the processor.

For both low-latency and high-throughput of event processing, Algorithm 3 does not rely on any interruption mechanisms to avoid trashing the instruction/data cache when switching to interrupt handlers, suffering from interrupt noise and interrupt handler control multiplexing and the overhead of context switching.

On more generic operating systems (Linux or Real-Time Operating System (RTOS)), this algorithm could use a preemptive and cooperative multi-thread (Line 19 in Algorithm 3). Nevertheless, in a high-performance environment, the OSs used on the targeted manycore have a simple run-to-completion multi-threading model in the matrix of Compute Clusters (CCs). The AOS library also provides another algorithm to let the programmer test, in a non-blocking way, whether or not the event is complete.

5.3.4 One-sided: RDMA and Remote Atomic Arbiters

The RDMA and *remote atomic* arbiters are high-performance algorithms that run on each Resource Manager (RM) of each CC of the MPPA®. The RDMA job arbiter is used to

Algorithm 3 Active Wait Event Algorithm (thread safe algorithm)

```
1: Input: event
2: value = get_event_check_value(event)
3: address = get_event_check_address(event)
4: while true do
5:    test_value = uncached load at address
6:    if Evaluate condition value with test_value then
7:       Read memory barrier /* Core & DMA coherence */
8:       Return success /* Exit */
9:    end if
10:   if is_event_has_pending_dma_eot(event) then
11:      eot = atomic load and clear on the End-of-Transfer (Eot) counter of the Rx Tag
12:      if eot > 0 then
13:         Atomic fetch add uncached eot at address
14:         Write memory barrier
15:         /* Force PEs to re-evaluate conditions */
16:         Broadcast notify to all PEs
17:      end if
18:   end if
19:   Idle PE                          // or OS yield possible
20: end while
```

process remote read transactions, that are the *Get* operations, initiated by the PEs. The remote atomic arbiter is used to process remote atomic transactions, initiated by the PEs.

Algorithm 4 serves the request sent by Algorithm 2 for the RDMA. The algorithm uses an efficient Round Robin (RR) arbitrations. The arbitrations are triggered on events (not interrupts) sent by the DMA NoC interface or inter-PE events. These arbiters process requests coming from the NoC or other intra-node PEs.

The RDMA job arbiter, in Algorithm 4, manages the execution of DMA jobs asynchronously. It selects the associated DMA micro-engine, configures the NoC route, writes the DMA micro-engine arguments, starts the DMA micro-engine, and updates the completion job ticket. No starvation is possible as one DMA micro-engine is dedicated to a single job FIFO.

An active message is an operation containing a set of instructions with operands. When the operation is performed, the active message job arbiter sends the result back to the initiator (if any) and updates the completion job ticket. The more complicated part of this software arbiter is that all active messages from an initiator are **ordered** with all outstanding RDMA writes of this initiator. For the initiator, all outstanding incoming RDMA transactions will complete before the posted remote atomic operation is processed.

5.3.5 Support of Eager Messages with Remote Queues

Classic two-sided *Send/Receive* operations have a significant overhead due to synchronizations between the sender and receiver nodes, and often require the use of temporary buffers as opposed to zero-copy communication. Besides, real-life implementations present significant challenges [Gor04] for simplicity, programmability, performance, and predictability.

As a primitive of the two-sided protocol, we select the remote queue operations described in [BCL+95], as it avoids the problems of classic message passing. First, it can be implemented as a simple message queues that are proven to be efficient for fine-grained

Algorithm 4 RDMA Engine Algorithm (this is a sequential task, communicating with distributed software)

```
1:  while true do
2:    List_Job_Fifo = Get pending job fifos
3:    for Job_Fifo in List_Job_Fifo do
4:      /* Transactions are ordered per fifo */
5:      DMA micro-engine = get associated micro-engine to Job_Fifo
6:      if DMA micro-engine is ready then
7:        Clear DMA micro-event event
8:        Read-memory barrier
9:        Read Job_Fifo[Currrent_Job_Fifo_Read_Ptr++] slot
10:       if New NoC route != Current Tx packet-shaper NoC route then
11:         Configure DMA Tx packet-shaper with New NoC route
12:       end if
13:       Write DMA micro-engine parameters & pointers
14:       Start DMA micro-engine
15:       /* For previous DMA job completions */
16:       Update array_dones_slot[current] completion counter
17:       Write memory barrier
18:       /* Force PEs to re-evaluate conditions */
19:       Broadcast notify to all PEs
20:       Current_Job_Fifo_Read_Ptr %= FIFO_SIZE
21:     end if
22:   end for
23:   Idle PE                                  // or OS yield
24: end while
```

control and coordination of distributed computations. Also known as eager messages, it allows low-latency for small messages. The maximum message size is usually given by an eager limit (usually a small number of bytes) which is implementation-specific [LJW+04]. From the sender point of view, the local buffer can be immediately reused. On the receive side, these eager messages can either arrive before or after the calling of the receiving primitive. Moreover, remote queues also apply to N-to-1 communication whenever atomicity of *enqueue* and *dequeue* operations can be ensured. The N-to-1 capability is essential on massively parallel systems that require run-time orchestration of activities (e.g., master/slave parallel pattern). Finally, remote queues enable efficient communications as they enable synchronization without introducing any locking mechanism from the programmer point of view.

5.3.6 Data Restructuring Support on RDMA Put/Get

Contiguous and strided copies are essential "geometries" in RDMA communications as seen in Figure 5.5. A strided transfer can have an offset between each contiguous data block either on *src* or *dst* buffers, or even both. An efficient RDMA API (and the underlying hardware) should be able to perform strided transfers with zero-copy, by automatically incrementing read and write DMA offset at no cost.

The implementation of the data restructuring feature uses the DMA micro-engine which runs a handwritten micro-code. The micro-code is a Kalray specific instruction set 2.3.3, written in TCL [O+89] where the pseudo code is given in Algorithm 5. The implemented

micro-code implements a stride-to-stride data transfers where the object size, the number of objects, the stride in bytes between the local object(s), and the stride in bytes between the remote object(s) are specified. 2D transfers are possible using a single iteration of Algorithm 5. However, 3D transfers require as many calls to Algorithm 5 as the depth of the cube to be sent.

Algorithm 5 sets the absolute remote offset of the targeted memory *window* (Line 3). Then a local address is set where the data to be sent shall be read (Line 4). Two loops are used to send the object of size *Object_Size* in bytes, using 8 bytes coalescing first, and 1 byte for the remaining. The remote offset and the local address are then updated (Lines 14 and 15), and the object is sent as many time as needed, using the Nb_Object input parameter. Once all objects are sent, Algorithm 5 sends the Eot command to the targeted memory *window* (Line 18) and notifies locally the RM that the operation is complete (Lines 19 and 20). The DMA micro-engine stops (Line 21) until the RM reuses it for another DMA job.

Algorithm 5 DMA Micro-engine Pseudo Code for Data Restructuring on a Window Memory Segment

```
1:  Input Local_Offset, Remote_Offset, Object_Size, Nb_Object, Local_Stride, Remote_Stride
2:  if Object_Size != 0 and Nb_Object != 0 then
3:    Send set absolute Remote_Offset command to the target DMA Rx Tag
4:    Set local read_pointer to Local_Offset (read pointer in local memory)
5:    for i in 1 .. Nb_Object do
6:      for j in 1 .. (Object_Size / 8) do
7:        Read and Send 8 bytes /* 8 bytes coalescing */
8:        Increment read_pointer of 8
9:      end for
10:     for j in 1 .. (Object_Size % 8) do
11:       Read and Send 1 bytes /* Send remaining bytes */
12:       Increment read_pointer of 1
13:     end for
14:     Send set relative Remote_Stride command to the target DMA Rx Tag
15:     Add relative Local_Stride to local read_pointer
16:   end for
17: end if
18: Send End-of-Transfer (Eot) command to the target DMA Rx Tag
19: Increment local DMA micro-engine event
20: Broadcast a notification to all local PEs of the CC
21: Stop DMA micro-engine
```

5.4 Use, Resource Allocation & Configurations

The AOS library performs a complex DMA NoC resource allocation at initialization time. Indeed, as the AOS engine enables relaxed one-sided operations on an MPPA® chip, a lot of software programmable hardware resources are used to deal with out-of-order completions and to remove software locks. The used resources are the packet-shapers, the DMA micro-cores, and DMA Rx Tags, all explained in Section 2.3.3. The packet-shaper is a hardware

functional unit that builds NoC packets to send them with a specific route over the NoC of the MPPA® processor.

The DMA NoC interface sharing of resources between other existing runtimes also needs to be managed. For instance, the Distributed Shared Memory (DSM) (Section 3.1.1), the OpenCL runtime and the low-level runtimes of the Kalray Neural Network (KANN) framework are important examples requiring interoperability tests.

Each Compute Cluster (CC) contains one DMA NoC interface that can access by *read* and *write* operations the local memory of the CC as explained in Section 2.3.3. On the Input/Output Subsystem (IO), 8 DMAs NoC interfaces are available for accessing the local memories of the IO (Shared Memory (SMEM)-Low and SMEM-High) and the external DDR off-chip memory by *read* and *write* operations.

As seen in Section 2.3.3, each DMA NoC interface has a limited range of hardware resources. To summarize, 8 packet-shapers, 256 DMA Rx Tag resources, and 8 DMA micro-engines are available in a single DMA NoC interface. As such, on a single MPPA® chip, 18 nodes are implemented, with 16 Compute Clusters (CCs) and 2 Input/Output Subsystems (IOs) where the one-sided and two-sided asynchronous communication features are implemented using the following set of hardware DMA NoC resources.

5.4.1 Resources Used for Enabling One-sided Operations

Managing the resource sharing to enable AOS on a two-sided NoC is complex. We highlight and show the purpose of the resources for each DMA NoC interface of a Node (CC or IO) of MPPA® in Table 5.1.

For One MPPA® Chip	IO DMA NoC Resources per Interface (8 Interfaces per IO)	CC DMA NoC Resources per Interface (1 Interface per CC)
Return of RDMA Get Operations	6 Rx Tags DDR-SMEM Low or SMEM High	18 Rx Tags
Incoming RDMA Put Operations	6 Rx Tags DDR 6 Rx Tags SMEM Low or SMEM High	18 Rx Tags
Get Job Fifo	1 Rx Tag	1 Rx Tag
Return of Active Message Data	6 Rx Tags DDR-SMEM Low or SMEM High	18 Rx Tags
Active Message Job FIFO	1 Rx Tag	1 Rx Tag
Return of Active Message Flow-control	1 Rx Tag	1 Rx Tag
Send Transaction Request	1 Packet-Shaper	1 Packet-Shaper
Serve Get Requests of other Nodes	1 Packet-Shaper 1 Micro-engine	1 Packet-Shaper 1 Micro-engine
Process the Locally Initiated Put	1 Packet-Shaper 1 Micro-engine	1 Packet-Shaper 1 Micro-engine
Serve Active Messages of other Nodes	1 Packet-Shaper	1 Packet-Shaper

Table 5.1 – *NoC Resources used by the AOS library for each of the Compute Cluster (CC) and each of the Input/Output Subsystem (IO) Composing an Entire MPPA® Processor*

One Rx Tag is required for each possible initiator Node for full-duplex (*Put/Get*) of the 18 Nodes (16 CCs and 2 IOs) within this Node. It is necessary to obtain a relaxed memory consistency model for the RDMA operations at multi-nodes level. The ordering and the memory consistency of the AOS library are explained in section 5.2.2.

On Input/Output Subsystems (IOs), 6 Rx Tags are required for optimizing the bandwidth of the data NoC targeting the DDR. 4 Rx Tags are used for the 4 rows or columns of Compute Cluster (CC) and 2 Rx Tags more are used for the Input/Output Subsystems (IOs) (IO-to-IO and loopback). Indeed, the routes used between IOs and CCs avoid turns by using only the columns or the rows of the NoC. This optimization is applied to avoid as much as possible the temporal sharing of data NoC links and limit NoC congestion.

5.4.2 Two-sided operations

The resource management for two-sided operations is simpler than the one-sided implementation, as the Kalray NoC is already two-sided. As such, Rx Tag resources are dynamically allocated in a software pool at AOS segment creation. At chip level, a centralized broker is in charge of the synchronizations and the forwarding of information such as the route and the allocated Rx Tag in the targeted remote queue. Centralized control and dynamically allocated resources have poor performances but this is not an issue since the segment creation is only perform once; therefore, it is not on the critical path of the application. The implementation of the remote atomic and the remote queue messages reuses the packet-shaper in Table 5.1 of the *Send Transaction Request*, for reducing the consumption of DMA NoC interface. Moreover, for non-atomic messages of the remote queues, it uses the packet-shaper and micro-engine of the one-sided Put operation, also visible in Table 5.1, namely *Process the Locally Initiated Put*.

5.4.3 Resources Necessary for AOS in a Compute Cluster

The resources used by AOS in a compute cluster are shown in Figure 5.6, which represents the DMA NoC interface configurations. One-sided operations (RDMA and remote atomics) operate on the entire OS and application memory space. The Kalray exokernel is protected from the OS and the applications by checking the configurations of the DMA NoC interface at initialization time. The micro-engines, PEs and the RM can read into the memory and write data in the packet-shapers. The packet-shapers then send data in the NoC with a pre-configured route and destination Tag. The DMA Rx writes in the memory the new incoming data to the window referenced by a Tag in the NoC header packet. On transfer completion, an Eot increments a DMA register counter related to the DMA Rx Tag as seen in Algorithm 3.

DMA NoC Interface Configurations Each hardware resource is configured at initialization time of the AOS engine or in the control path of the AOS engine. **Indeed the configuration of hardware resources is not in the data transfer path for performance.** The following configuration parameters need to be set by the software operating in the AOS initialization function, for each of the used DMA NoC interface resources. The resources configurations are listed below:

- **DMA Packet-shapers** [0..7]
  - Destination DMA Rx Tag in the range [0..255]

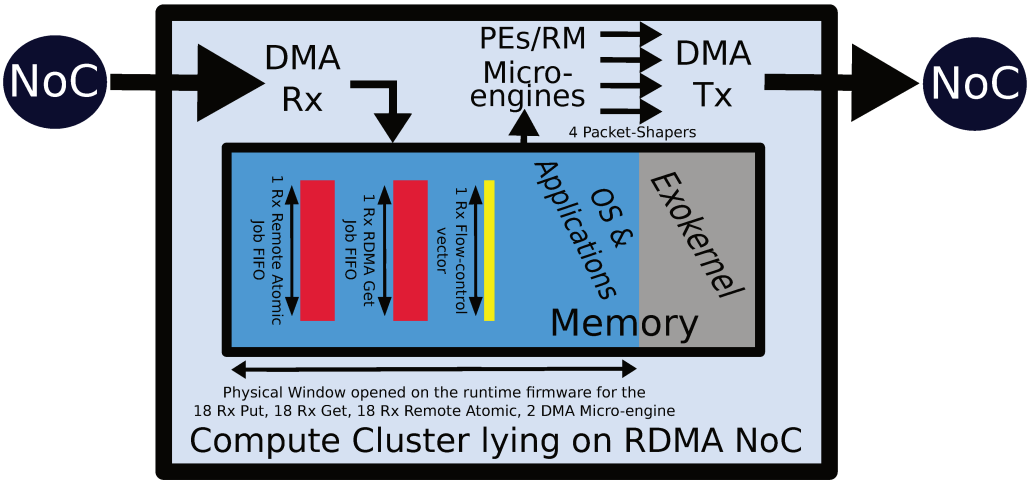


Figure 5.6 – *Architecture of AOS in a Compute Cluster*

  - Route: Unicast NoC Route using X-Y Routing. The route is a sequence of North, South, East, and West commands encoded respectively in binary: 00, 01, 10, 11. The route stops when the current direction goes to the same direction that the packet came.
  - First Direction: North/South/East/West/Loop-back. The Loop-back is encoded as 100.
  - Bandwidth Limiter: Set to best effort.
  - Minimal / Maximal Payload: 4-flit / 32-flit. 1 flit is 4 bytes which are the NoC bus width (32-bit).

- **DMA Rx Tags** [0..255]
  - Activation: Enables the DMA Rx Tag, NoC packets are dropped overwise.
  - Buffer Base: Physical start address of the window where the DMA can write.
  - Buffer Size: Size in bytes of the physically contiguous window.
  - Mode: Increment the Event counter and notify the processors when receiving an End-of-Transfer (Eot) command.
  - Item and Event counters: Set to zero at initialization.
  - Notification Vector: A vector of bit-mask indexing the processors to which an event shall be raised when an Eot command is received.

- **DMA Micro-engine** [0..7]
  - Buffer Base: Physical start address of the window where the DMA can read.
  - Buffer Size: Size in bytes of the physically contiguous window.
  - Micro-code start address: The start address of the code of the micro-engines, explained in Algorithm 5.
  - Event counter: Set to zero at initialization.
  - Notification Vector: A vector of bit-mask selecting the PEs to which an event shall be raised when the micro-engine completes its job. In the AOS engine, the micro-code increments this Event counter Line 20 in Algorithm 5.

When a notification from a DMA Rx Tag or a DMA micro-engine occurs, the notified processors go out of the idle state, or future execution of the idle instruction will not idle the PE. A clear of the PE wake-up information should be done by software hereafter.

5.5 Performance, Results: Latency & Throughput

We use a multi-CCs execution model with a low-level POSIX-like (Pthreads) environment for benchmarking. All measures were made onto an MPPA® operating at 500 MHz with one or two 1066 MHz DDR3. Each DDR3 bus size is 64-bit wide which leads to a theoretical and maximum memory bandwidth of 8.5 GB/s, and 17.0 GB/s using 2-DDRs. The NoC is 32-bit wide operates at 500 MHz too; therefore, it provides a bandwidth of up to 2.0 GB/s per link. The SMEM of the CC has 1 NoC link providing 2.0 GB/s per link direction. However, we use a typical data NoC payload packet size of 32 flits with a header of 2-flits for a total typical packet size of 34 flits. Thus, it leads to a maximum efficient data transfers throughput of $2 * (32/34)$, which gives 1.88 GB/s full-duplex. The memory throughput is defined as the memory bandwidth on which the node(s) or processor(s) are reading or writing. The latency is defined as the time between the initiation and the completion of a transaction; thus, it will depend on the size of the transaction.

5.5.1 Memory Throughput

Figure 5.7 and Figure 5.8 shows both DDR(s) and inter-cluster SMEM reads and writes, respectively called *Gets*, and *Puts*. The throughput is measured onto the memory on which the CCs are reading and writing. The size of the RDMA transactions in abscissa and the number of CCs are varying, showing different saturation points. All throughput benchmarks were with asynchronous RDMA transactions to saturate the software in charge of configuring the DMA NoC interfaces. Thus software flow-control is heavily used to prevent the local or remote FIFOs from getting corrupted.

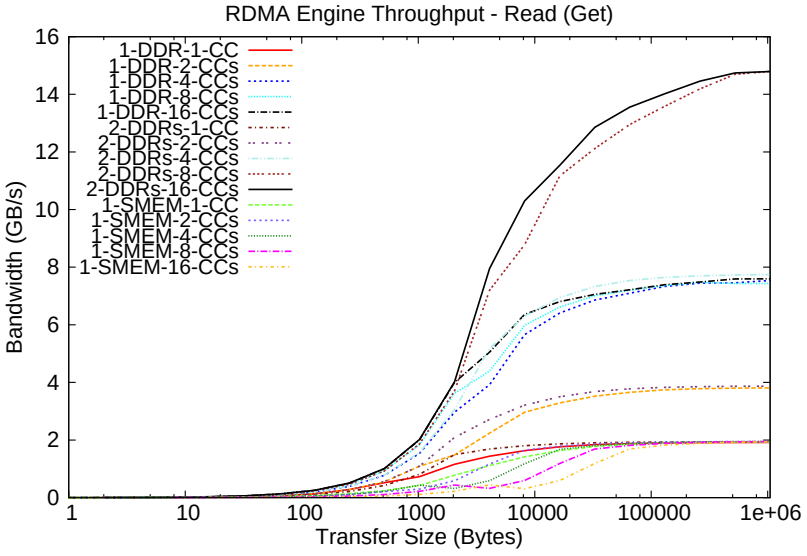


Figure 5.7 – *RDMA Get (Read) Throughput GB/s (Asynchronous)*

Firstly, for DDR memory accesses, we achieve more than 50% of maximum theoretical throughput for data transfers larger than 4 KB and 94% for 32 KB in all topologies. Secondly, it can be noticed that RDMA *puts* are better than *gets*. It is due to remote

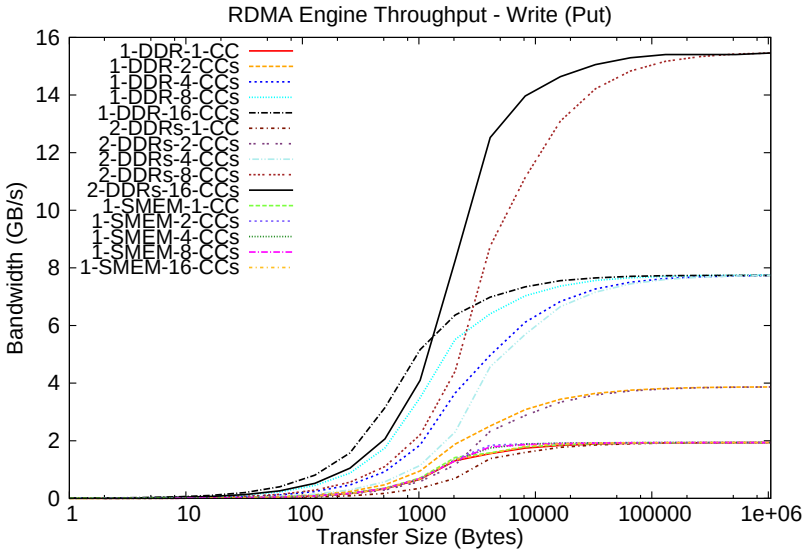


Figure 5.8 – *RDMA Put (Write) Throughput GB/s (Asynchronous)*

server contention which is the point of serialization for the configuration of the DMA interfaces. Indeed, on the software point of view, outstanding *puts* only rely on local flow-control whereas outstanding *gets* rely on remote flow-control. Remote flow-control is more complicated as it requires more software interactions with the DMA NoC interface. We measure that our software implementation of RDMA support reaches more than 70% of the peak hardware throughput for a contiguous data NoC stream size larger than 8 KB. To conclude, the RDMA throughput provides the user application with efficient use of the hardware, when having data stream size greater equal than 8 KB, and manages complex flow-control mechanisms. Providing the programmer with performance and implicit flow-control eases the implementation of explicit communications.

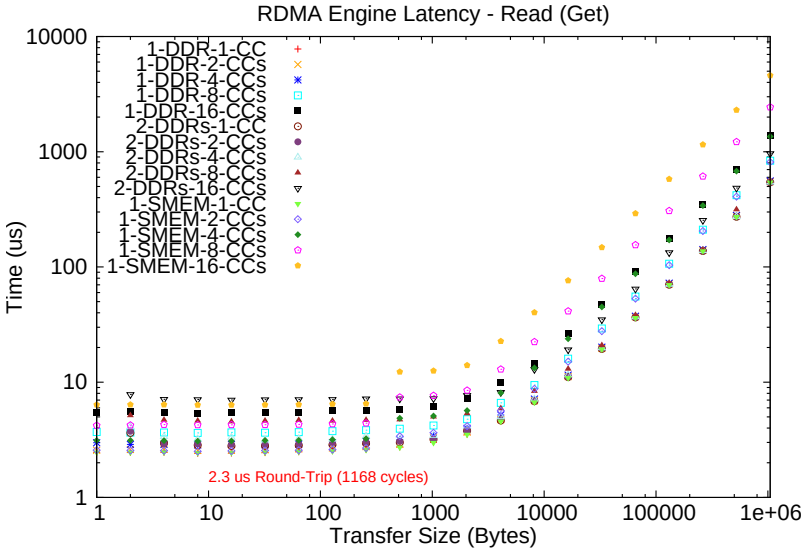


Figure 5.9 – *RDMA Get (Read) Latency µs (Blocking)*

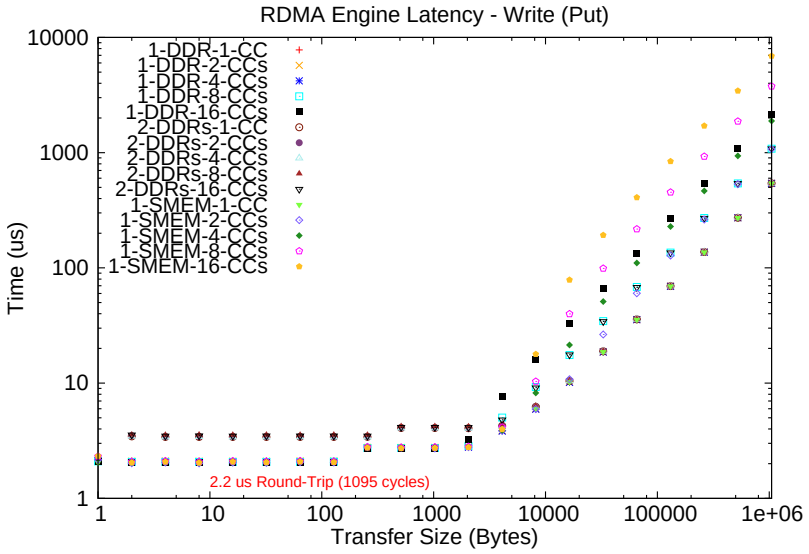


Figure 5.10 – *RDMA Put (Write) Latency µs (Blocking)*

5.5.2 Memory Latency

The software in charge of configuring the DMA NoC interface introduces some latency. Highly-coupled parallel software often leads to poor performance onto massively parallel architectures. Therefore the transaction latency on such architectures is critical when dealing with complex data dependence patterns that imply inter-clusters communications (e.g., low-latency 6-steps Fast Fourier Transform (FFT) or low-latency CNN inference). However, depending on the spatial and temporal memory locality, it is not always possible; and thus, the latency of the transactions becomes important. We model the total round-trip latency of the RDMA software/hardware engines when there are neither congestion nor user/kernel interruptions. $TT(B)$ is the *Time to Transmit B bytes* and is given by: $TT(B) = IPT + HLT + SPT + B/3.76 + CT$. Let IPT be the *Initiator Processing Time* described in Algorithm 2, SPT the *Server Processing Time* explained in 5.3.4, HLT the *Hardware Latency Time* for the NoC link/router and micro-engine memory accesses, CT the *Completion Time* described in Algorithm 3 and the transfer time $B/3.76$ with B the number of bytes to transfer. 3.76 bytes per cycles is the efficient data transfer bandwidth considering a NoC header of 2 flits with a payload of 32 flits. Typical cost in cycles are respectively: $TT(B) = 500 + 100 + 300 + B/3.76 + 200 = 1100 + B/3.76$.

Figure 5.9 and Figure 5.10 show round-trip latency using different compute matrix geometries that are reading or writing into DDR(s) or one SMEM. The minimum latency is $2.2\mu s$. When transfer sizes are greater than 10 KB, we observe the point of rupture. After the rupture point, this software latency becomes negligible compared to the latency the of DMA micro-engine transfer. When there is no contention, for instance, as observed in curve *1-DDR-4-CCs*, we have precisely a curve derivative of 3.76 bytes per cycle after this rupture point. Moreover, after this rupture point, the latency is impacted by the bandwidth of the external memory, that is either the DDR or the SMEM of the CC.

5.5.3 Network-on-Chip Scalability

A strength of NoC-based manycore processors is the ability to scale on non-interferent inter-node data transfers. Table 5.2 shows the internal compute matrix NoC bandwidth using different matrix sizes and stream sizes. The peak input-output throughput of the 16

CCs is given by $2*16*3.76$ bytes per cycles. Operating at 500 MHz, it provides 60.2 GB/s peak bandwidth. Our new RDMA engine can reach more than 88% of peak performance for inter-node transfers with a size of 16 KB.

Transfer Sizes / Nb Cluster(s)	1KB	4 KB	16KB	64KB	256 KB
1	0.7	2.7	3.5	3.8	3.8
4	2.7	11.0	14.2	15.2	15.4
8	5.5	21.8	28.3	30.5	30.8
16	10.7	42.8	56.3	60.2	60.2

Table 5.2 – *NoC Bandwidth of the Compute Matrix in GB/s*

The communication patterns are the following: each Compute Cluster (CC) initiates an RDMA *Put* to a neighbor using NoC routes that do not overlap between each other at runtime. When using 1 CC, we use the loopback feature of the NoC interface. No NoC link sharing or point of serialization occur, but the share of the SMEMs for the DMA NoC interface reads and writes. The SMEM is a multi-bank interleaved memory of 16 banks, and each bank can sustain 8 bytes per cycle; therefore, providing a bandwidth of 64 GB/s, this is not the bottleneck in our measurements.

5.5.4 Remote Atomics Performance

We benchmark the latency of the active message engines as they are used for synchronization and reduction operations. Figure 5.11 shows the latency on different matrix size for both asynchronous and blocking calls. In abscissa, we show the number of initiator CCs that are targeting either in the spread or centralized mode the 16 CCs of the manycore. Spread mode means that all initiators change their target CC each time they are sending a request. They all target different CC. It can be understood as a scatter mode with no overloading on receivers. It is quite well load-balanced, and the best performance is expected. Centralized mode means that all initiators target simultaneously the same node, thus overload this node by increasing request processing. It is the case of the reduction pattern for instance. It aims to measure the worst case of all possible active message scheduling schemes at execution.

The best case initiator latency on a posted operation is 230 cycles ($418ns$), for instance, *postadd*. The round-trip latency for the completion of a *fetchadd* operation is 1109 cycles, $2.2\mu s$. Lots of conflicts occur when the 256 PEs send requests to the same cluster. Curve "Async-Centralized-16-PEs" with 16 CCs shows such conflicts as this has a higher execution time. In such a configuration, the N-to-N flow-control is generating much traffic to avoid the corruption of software job FIFOs. The implementation can sustain such contention; however, the latency explodes. An execution time of $17.5\mu s$ in asynchronous mode is measured, whereas an execution time of less than a $1\mu s$ is measured when there is no congestion.

5.5.5 Remote Queue Throughput

Remote queues provide an elementary support of two-sided communications for small low-latency atomic messages (1-to-1 and N-to-1). Regarding the benchmark conditions, each CC has a queue where the IO is sending messages to the CCs. Then the CCs gets this message, sent by the IO, and responds using a remote queue message in N-to-1 mode. All

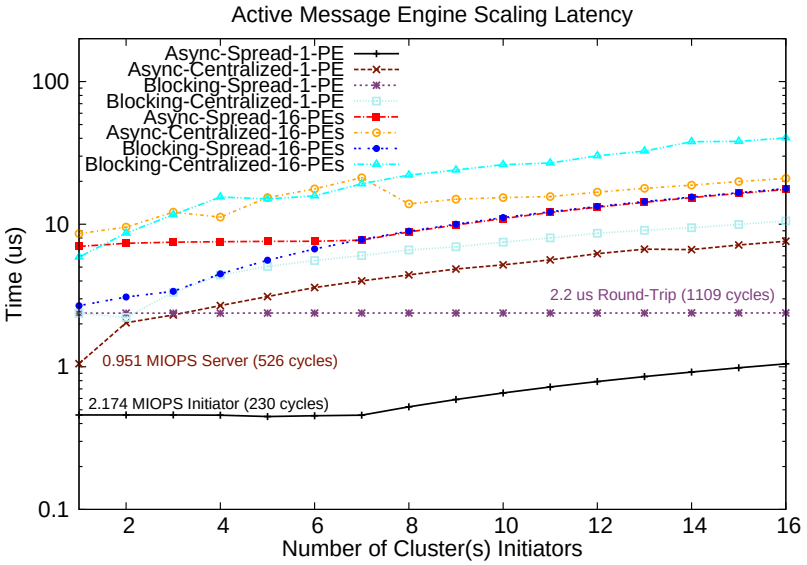


Figure 5.11 – *Active Message Latency*

CCs are running concurrently; therefore, we use the N-to-1 feature for data NoC packet atomicity. The hardware serializes received messages using this N-to-1 feature on the IO side. Table 5.3 shows IOPS of one RM of the IO that is receiving a request from a CC and is sending back a new job command to the responding CC. The benchmarks were carried out using different data NoC packet sizes (Bytes) without any batching.

Packet Size in Bytes / Nb Cluster(s)	16 B	32 B	64 B	128 B	248 B
1	675	670	600	425	350
2 to 16	740	725	725	575	550

Table 5.3 – *Performance of the Remote Queues in Kilo IOPS*

For small messages, the results show that a simple double-buffering using 2 CCs saturates the number of IOPS for one IO master PE. For messages bigger equal than 128 bytes, the IOPS drops, as the time spent by the IO to send the message increases linearly with the message size.

Such a communication pattern is crucial as it is used in offloading. Therefore, we provide the limitation of such implementation when fine-grained parallelism is required by an offloaded application, where the control is done by a host processor. In our case, the IO is considered as a host processor, that is offloading computations onto the compute matrix (though job queue commands).

5.6 Advanced Asynchronous One-Sided Support

This section presents the use of the Asynchronous One-Sided (AOS) in the Kalray Linux Kernel ported onto the Input/Output Subsystem (IO) in Section 5.6.1. Also, we propose the use of AOS in the OpenCL implementation presented in Section 5.6.2.

5.6.1 Support in the Linux Kernel

In the latest MPPA® AccessCore toolchain release (> 3.0), Linux, explained in Section 2.4.1, aims to be the primary Operating System (OS) running on the Input/Output Subsystem (IO) of MPPA®. Indeed a port of the Linux OS has been performed onto 4-core of an Input/Output Subsystem (IO) of MPPA®. It is possible thanks to the shared L1 data cache of IOs as the Linux OS requires data cache coherency.

AOS was initially designed to operate at a bare-metal level for high-performance. As Linux is a rich OS, it is mandatory to have a driver to access the AOS services. A driver provides clean isolation between the user-land and kernel land, manages concurrent accesses to the real hardware resources and makes it possible for the Linux kernel to check user commands from Input/Output Control (IOCTL) in such a way that it will not corrupt the Linux kernel. The memory swap feature of Linux is not activated in the Kalray port onto the Input/Output Subsystem (IO) of MPPA®. Therefore, in collaboration with two engineers, Guillaume Thouvenin (for the driver skeleton and tests) and Benjamin Mugnier (user library and tests), the AOS engines were ported in the kernel space of the Kalray Linux.

The driver skeleton consists of the creation of a new driver in the Kalray Linux, the factorization of some of the IOCTLs, finding the proper sequence for initializing the AOS driver at Linux boot time and the deployment of the AOS micro-firmware for efficient scheduling of AOS jobs as in Algorithm 4. The user-space library performs the required IOCTLs to invoke in kernel space (in the driver) the AOS services.

AOS Engines in Linux Kernel Space Most of the work consisted in porting in the Linux kernel space the one-sided and two-sided operations of AOS namely, the RDMA *Put/Get* operations, the remote queues and the remote atomic operations. The biggest challenges were to deal with the virtual memory address space and the Kalray heterogeneous Linux memory map. The DMA NoC interface does not support IOMemory Management Unit (MMU). The translation between the virtual memory and physical memory for initiating DMA NoC data transfers has to be managed by software. The translation between the virtual and physical address space is done in the kernel space using the *virt_to_phys* function once the address virtual address has been retrieved in the kernel space. Also, a Continuous Memory Allocator (CMA) allocator is provided for dealing with continuous memory space for efficient big memory buffer data transfers. Otherwise, buffers are split across the paged virtual memory space and a lot a software overhead is added as there is no IOMMU support in the Kalray DMA NoC interface.

5.6.2 Extensions and Support of the Standard async_work_group_copy() in the Kalray OpenCL

As seen in section 4.1.3, the OpenCL standard defines primitives providing explicit memory accesses between the `__global` memory address space and the `__local` memory address space. Our new designed AOS engine provides the necessary back-end to implement those standard primitives of OpenCL efficiently. Indeed, the AOS library implements out-of-the-box the required primitives to support standard OpenCL functions: “async_work_group_copy()” and “async_work_group_strided_copy()”.

The Kalray’s OpenCL runtime operates at low-level (bare-hypervised), as such, most of the work was to initialize the AOS engine in the initialization part of the OpenCL runtime. The most complicated part was the interoperability of the AOS engines and the Distributed Shared Memory (DSM) systems, regarding resource sharing and boot synchronizations. In

collaboration with Romaric Jodin, as the AOS primitives are part of the low-level runtime, we also had to expose these new primitives, and allow them to be called from OpenCL-C kernel. Indeed, OpenCL kernels that are deployed inside the CCs are loaded dynamically; thus, the relocation is performed using a dynamic overlay.

Finally, the new support of asynchronous copies in the Kalray OpenCL is now provided with extensions such as 2D and 3D asynchronous buffer copies. Moreover, batched and arbitrary stride-to-stride transfers are also available and part of the Kalray extension. As explained in Section 5.3.6, these functions provide zero-copy asynchronous memory transfers. They are fundamental for performance optimizations on a manycore architectures where the time to access the main memory is high.

Today, async_work_group_copy primitives are the state-of-the-art of the optimization of OpenCL applications onto MPPA®. Indeed, the asynchronous work-group-copy primitives allow the application developer to pre-fetch data from the `__global` memory space to the `__local` memory space to overlap the computations with the communications. The async_work_group_copy primitives give the ability for the developer to bypass the Distributed Shared Memory (DSM) system which yields poor performances when memory access patterns are sparse at the L2 cache geometry. Also, as the Distributed Shared Memory (DSM) system is *write allocate* at L2 level (contrary to the L1, see Section 4.5.1), it reduces by half the consumed memory bandwidth on write-only memory buffers. As such, the new free bandwidth can be used for other memory transactions, either by *Load/Store* using the Distributed Shared Memory (DSM) or explicitly using the new asynchronous work-group-copies.

5.7 Conclusion

We present the design and illustrate the advantages of a one-sided asynchronous (AOS) communications and synchronizations programming library for the Kalray MPPA® processor. The motivation is to apply to this and related CPU-based manycore processors the established principles of one-sided communications libraries of supercomputers, in particular: the Cray SHMEM library, the PNNL ARMCI library, and the MPI-2 one-sided communication API subset. The main difference between these communication libraries and the proposed AOS library is that a supercomputer has a symmetric architecture, where the compute nodes are identical, and the working memory is composed of the union of the compute node local memories. Similar to Infiniband low-level API, the AOS programming library supports the Read/Write, *Put/Get* and Remote Atomics protocols, but these have been designed around the capabilities of an RDMA-capable NoC.

One-sided asynchronous operations are proven to be highly efficient on the MPPA® NoC thanks to relaxed ordering and easier to use as the initiator is a master on the target memories. Indeed, one-sided communications do not require strict matching whereas the *Send/Receive* operations do in the two-sided protocol. Our software implementation can sustain more than 70% of the hardware peak throughput when using RDMA *Put-Get* engines for data transfer sizes greater equal than 8 KB. However, managing resource sharing, flow-control, arbitration and notifications in software has limitations in terms of latency. Based on this implementation and its results, the forthcoming 3rd-generation MPPA® processor will include hardware accelerations for these critical functions. Very low-latency transfers and peak throughput on small transactions will be obtained, along with respecting important ordering properties for the global memory consistency.

This communications and synchronizations runtime not only provides the user with the best performance on this hardware. It also hides the DMA NoC interface complexity to the

programmer without any penalty in efficiency. As explained in Section 5.1, multiple challenges are solved such as complex N-to-N flow control, resource sharing, abstraction of the target processor architecture, and implicit synchronization at memory segment creation. These provide significant ease for the programmer.

The AOS communication and synchronization library [HdDdMH17] has been deployed in production in the MPPA® AccessCore toolchain since 2017. It supports diverse application programming of the MPPA® processor for a range of low-level and in high-level programming environments such as OpenCL, OpenMP, OpenVX (Chapter 9), execution back-ends for static (Chapter 7), and dynamic dataflow (Chapter 8), programming models. The AOS library is also used by Kalray's optimized application libraries like the BLIS [VZVDG15] framework for high-performance Basic Linear Algebra Subprograms (BLAS), and it is targeted by code generators such as for CNN inference. In 2018, AOS was also deployed in Kalray's networking solutions as one of the foundations for reaching IOPS performance for NVMe use-cases [BYY+16].

CHAPTER 6

A Highly Efficient Multi-threading Runtime

In this chapter, we introduce and provide a new runtime for managing at low-level the threads running on a multi-core processor with shared local memory. The performance of such a multi-threading runtime is crucial, especially when fine-grained parallelism is required. Our New Multi-Threading Runtime (NMTR) is based on lock-free mechanisms [Bar93] [MP92], known to be effective for the implementation of Operating System (OS) kernels, and efficient parallel implementations. In this chapter, we use these techniques to implement a lightweight runtime, adapted to the targeted architecture fitted with shared local memories. These shared local memories support atomic operations in hardware as seen in Section 4.5.4. We explain in this chapter the various implementations of functions and mechanisms operating on the multi-core Central Processing Units (CPUs) of the Multi-Purpose Processor Array (MPPA)® processor, namely the 2 Input/Output Subsystems (IOs) and the 16 Compute Clusters (CCs). This contribution provides the programmer of the multi-cores with less scheduling and synchronization overheads compared to the state-of-the-art multi-threading runtimes.

The chapter is organized as follows. We present in Section 6.1 the issues to be solved to make the toolchain of the targeted manycore compatible with multi-threading, and the limitations of the target processor when standard codes are executed. Section 6.2 presents in details the new multi-threading runtime and gives the most essential algorithms that were designed to enable efficient multi-threading. The synchronization primitives used by the runtime are presented in Section 6.3, and the multi-thread cooperative scheduler is explained in Section 6.4. In Section 6.5, we provide keys to enable OpenMP based on the GNU Compiler Collection (GCC) libgomp runtime back-end and optimize it. We reduce the used memory of the OpenMP runtime library when its use is ended at the application level. The MPPA® OpenMP runtime of GCC libgomp depends on our new multi-threading runtime. Section 6.6 explains a new multi-threading mode based on automatic thread yielding onto Direct Memory Access (DMA) job event completion. Section 6.7 presents experimental results evaluating the effect of our contributions on the runtime performances and gives hints to enhance this new multi-threading runtime on MPPA®.

6.1 Controlling and Enabling Threads for a Non-Coherent Multi-core CPU

The threads are usually managed by a master thread or any other threads having access to the credential or reference to these thread resources. The control of threads is explained in Section 3.1.2, using the Pthread standard. Our New Multi-Threading Runtime (NMTR) provides the most commonly used thread management functions and enables efficient and generic fined-grained multi-threading (e.g., thread creation, synchronization, and semaphores). The internal low-level functions of the runtime are then exposed using the standard Pthread Application Programming Interface (API) that we implemented.

In the targeted toolchain, the Pthread function definitions are given in the *newlib* open-source library [new]. The *newlib* is usually built before the final compiler for the targeted machine. As such, *newlib* provides the definition of most runtime functions or services such as *syscalls*, *input-output operations*, *errno*, memory allocator and the *Pthread* API definition (a bunch of *.h* standard files). Our NMTR then implements the Pthread API.

The support of Pthread functions makes it possible to run most Linux multi-threaded applications on a single multi-core Central Processing Unit (CPU) of the Multi-Purpose Processor Array (MPPA)® which is either an Input/Output Subsystem (IO) or a Compute Cluster (CC). However, out-of-the box multi-thread programs commonly assume a coherent memory hierarchy. As the Compute Clusters (CCs) of the MPPA® processor do not support a coherent memory hierarchy, the software runtime performs cache management operations at synchronization points, providing software coherency for most multi-threaded programs. However, an off-the-shelf lock-free multi-threaded software, written for performance, will not work as it usually requires a coherent memory hierarchy.

6.2 Implementation of the New Multi-Threading Runtime

In this section, we present the diverse states of the thread resource in Section 6.2.1. For performance, the management of false positive, and masked interrupts are explained in Section 6.2.2. Section 6.2.3 presents the basic primitives that are used to control the thread execution.

6.2.1 Logical Thread States

Each executing thread is in a state or a transition at some point during its execution. At the start, threads are pre-booted by the runtime and placed in the *idle* state. Figure 6.1 shows the typical states of a thread in our NMTR. The states of threads represented in Figure 6.1 are a coarse-grained simplification of the actual thread implementation. Details of the actual thread states and their implementations are explained in Section 6.2.

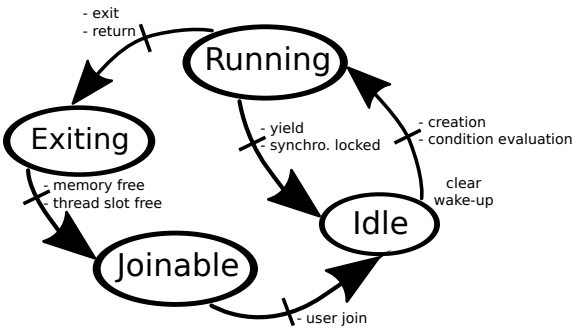


Figure 6.1 – *Specific States & Transitions of Threads in the NMTR*

6.2.2 Dealing with System False Positives and Masked Interrupts

In this section, difficulties to achieve high-performance implementation for our NMTR are exposed. The problem of interrupts is explained and hints to overcome this issue are suggested. Solutions are then adapted to our proposed NMTR.

An interrupt is an external event that stops the current execution of the running thread, to make it switch to another function handler. An interrupt handler is usually a short procedure. However, the Operating System (OS) must perform a full context switch. When the interrupt handler completes, the initial thread is resumed.

Interrupts Issues

Interrupt-based systems are widely used in computer systems for dealing with external events. As such, interrupts trigger actions, possibly when one or several events occur at some point during the execution. On a large system, many events can happen during the execution. Therefore, dealing with interrupts usually leads to poor performance, because of the following four issues:

1. Instruction and data cache stalls (misses) due to a loss of locality as the core switches to other handlers.

2. Core context switch. Additional latency is due to the spilling of the entire register file memory and reloading another context from memory. Depending on the operating system, multiple switches between the user-space and kernel space must take place.

3. The aggregation of several interrupts in the same interrupt handler that requires additional software control. The core needs to find out what happened: overhead.

4. Thread management overhead. Each thread may be blocked on one or several conditions which must be checked by the scheduler: more overhead.

In high-performance computing, interrupts are known to be the wrong approach for the implementation of effective control of fined-grained external events [CAR14]. For latency optimization, experts would instead use polling to react faster. Furthermore, in the Linux kernel space, such polling mechanisms with disabled interrupts are used to make low-latency time reaction possible. However, limitations can be reached on such “best effort” implementations regarding real-time computing.

System Masked Interrupts & False Positive

Interrupts can be masked; if this is the case, nothing happens for the core when the interruption occurs. Indeed, the core does not switch to an interrupt handler when the interruption occurs. Masked interrupts consist in waking the core up if it was idled when an external event is generated by one of the following initiators: Direct Memory Access (DMA), Peripheral Component Interconnect Express (PCIE), Input/Output Subsystem (IO)-Memory Management Unit (MMU) or any other IO device. Instead of switching to an interrupt handler, the core polls the memory or the peripherals for an event. In most OSs, such mechanisms can only be performed in kernel space as they require atomicity, ordering, and the control data must be centralized.

Another issue is the handling of false positives. False positives may happen when cores (Processing Elements (PEs)) share the same events of external resources. In this case, when the cores are awakened, they must be robust to data races in control structures and manage the hardware resource sharing correctly. For performance, locks are prohibited, and we use states (shared data) in memory to perform decisions. These states in memory are usually multiple readers and a single writer. At worst, a lock-free atomic operation is used when multiple writers need to perform atomic updates on some particular data structures in memory. Therefore, the main requirements are always to guarantee atomicity and commit order regarding the handling of the completion of external events. Also, the aggregation of events must be performed to avoid the loss of events. The technique consists of processing every pending request before deciding to idle the core, only in this order. The core goes out of idle state when new external events occur.

6.2.3 Thread Control

As seen in Section 3.1.2, the `pthread_|create|join| yield|exit|` primitives are sufficient to control the threads. We implemented these functions on the user-space side to make them light and efficient. Indeed, these functions use neither *syscalls* nor *interrupts* which can have a significant overhead.

All of these functions are lock-free. On massively parallel architectures, an efficient low-level software is achieved using lock-free software implementation when possible. Indeed, when a lock mechanism is used on a congestion state, the latency in cycles of the critical section is given by the number of cores multiplied by the number of cycles of the critical section itself and the latency to take and release the lock when it is available.

The thread object is an opaque pointer from the programmer point of view. Opaque pointers hide the implementation of the object. Many software libraries use such a technique to hide the manipulated objects from the programmers, like the *newlib* [new] or most APIs of the well-known Khronos consortium.

We provide the pseudo-code of our implementation of the NMTR in Algorithms 6, 8, and 9. Behavioral and implementation details for all the referenced algorithms are also given.

Thread Creation

The creation of threads is a fundamental operation of the NMTR. This procedure let the programmer execute a function handler onto a PE, for parallel computing. In our NMTR, the thread creation function is always called at the user level, providing competitive performance.

The entry point of the *main* program is executed on the first PE of each multi-core CPU of MPPA®, which is in our case the CC or the IO. As explained in Section 2.3.2, CCs have 16 PEs, and IOs have 4 Resource Managers (RMs), denoted *NB_CORES*.

On each PE, it is currently possible to execute up to 4 software threads, denoted *NB_SOFT*. There is neither preemption mechanism nor thread migration, meaning that the threads are cooperatively executed on the PE to which it is assigned. The cooperative execution means that there is no preemptive scheduling. The PE releases the running thread only on some specific thread management functions that are explained below in this chapter.

The pseudo code for creating threads from any cores is given in Algorithm 6. This algorithm takes as inputs the address of the function to be executed, the address of the data given to this function, the core ID in the range [0 ... (*NB_CORES*-1)], and a selection mode that either specifies the core to which the function should be executed or choose the core with fewer threads (AUTOMATIC_SELECTION).

The output of the algorithm is an opaque thread object that contains:

- The ID of the core to which the thread is pinned.
- The ID of the software thread running on this core.
- The state of the thread, such as idle, running, exiting or joinable.
- The ID of a slot in a pre-allocated array to store some internal information related to the thread, such as the stack address, the Thread Local Storage (TLS) address, the user callback address, and the architectural states of the k1-Very Long Instruction Word (VLIW) core.

When the *Selection_Mode* input is set to automatic (*AUTOMATIC_SELECTION*), the core calling the thread creation algorithm selects the first available thread slot, in order, concerning the number of hardware core. A thread slot is an idle soft-thread resource on a hardware PE that is waiting to execute a function handler.

The main thread operates on core 0 slot 0, which is the entry-point of the program running on the multi-core CPU. When creating threads in automatic mode, this main thread selects new thread resources to be placed on a core in the following order: Slot 0 core [1 ... (*NB_CORES*-1)], Slot 1 core [0 ... (*NB_CORES*-1)], Slot 2 core [0 ... (*NB_CORES*-1)], Slot 3 core [0 ... (*NB_CORES*-1)]. A very efficient implementation is to use the count-trailing-zero[1] instruction followed by a compare-and-swap[2] to update atomically and concurrently the thread array element of bits, in a lock-free manner (see Line 15 of Algorithm 6).

The streaming loads (uncached) and atomic operations (uncached) are explained in Section 4.5.4 for the k1-VLIW core. In the described algorithms, the compare-and-swap atomic memory operation has the following semantic: *boolean = compare-and-swap(old_value, new_value)*. For the implementation of both the IO and the CC, the k1-VLIW atomics have the same effect as uncached memory operations, as explained in Section 4.5.4.

Thread Yield

The yield operation let the programmer put the calling thread in idle state. The current thread is placed in idle state and ready to be rescheduled once other threads operating on

[1]Standard GNU Compiler Collection (GCC) builtin: __builtin_ctz
[2]Standard GCC atomic: __atomic_compare_exchange

Algorithm 6 Thread Creation Algorithm (this is a thread safe algorithm).

```
1: init: 64bit_slot_mask_addr = ∼(((∼0ULL) << (NB_SOFT*NB_CORES)) | 1)
   64bit_slot_obj_alloc_addr = ∼0ULL
2: Input: Function_data_addr, Function_addr, Selection_Mode, Core_ID
3: Output: Thread_Object { Core_ID, Thread_ID, Core_State,
   Static_Memory_Slot_ID }
4: Static Thread_Object_Table[NB_CORES * NB_SOFT]
5: slot_mask = load-uncached(&64bit_slot_mask_addr)
6: slot_object_alloc_mask = load-uncached(&64bit_slot_obj_alloc_addr)
7: while true do
8:    if slot_mask == 0 then
9:       Return failure, no thread slot available
10:   end if
11:   if Selection_Mode != AUTOMATIC_SELECTION then
12:      slot_mask &= ((1ULL<<(3*NB_CORES)) | (1ULL<<(2*NB_CORES)) |
         (1ULL<<NB_CORES) | (1ULL)) << Core_ID
13:   end if
14:   Thread_ID = count-trailing-zero(slot_mask)
15:   if compare-and-swap(&64bit_slot_mask_addr, slot_mask,
      slot_mask & (∼1ULL<<Thread_ID)) then
16:      break
17:   end if
18:   slot_mask = load-uncached(&64bit_slot_mask_addr) /* Let's retry */
19: end while
20: while true do
21:   if slot_object_alloc_mask == 0 then
22:      /* Always success, enough memory and atomic */
23:      Thread_Object = dynamic alloc heap with libc
24:      break
25:   else
26:      Slot_object_alloc_ID = count-trailing-zero(slot_object_alloc_mask)
27:      if compare-and-swap(&64bit_slot_obj_alloc_addr,
         slot_object_alloc_mask,
         slot_object_alloc_mask & (∼1ULL<<Slot_object_alloc_ID)) then
28:         break
29:      end if
30:   end if
31:   slot_object_alloc_mask = load-uncached(&64bit_slot_obj_alloc_addr)
32: end while
33: Thread_Object = { .Core_ID = Thread_ID % NB_CORES,
   .Thread_ID = Thread_ID, .Core_State = Created,
   .Static_Memory_Slot_ID = Slot_object_alloc_ID }
34: Streaming-store to write core context in the Thread_ID slot (Function_data_addr,
   Function_addr, Thread_Local_Storage_addr, Thread_Stack_addr)
35: Write memory barrier /* Stalls the core until all write accesses are completed */
36: Streaming-store to write Runnable state to the Thread_ID slot
37: Write memory barrier /* Stalls the core until all write accesses are completed */
38: Broadcast notify to all PEs
39: Return success
```

the same core are idled, at the exception of the *main* thread to prevent deadlock. Also, if no other software threads are running on the core, it means that only one thread is currently running on the hardware core, thus, the yield operation simply goes to the scheduler, and the scheduler immediately reschedules this yielding thread.

The yield operation is given in Algorithm 7. Before calling the cooperative scheduler, all callee-saved registers, plus other registers such as the stack pointer, the return address, the frame pointer for the debugger, the Thread Local Storage (TLS) pointer (__thread attribute to variables as seen in Section 2.4.2) and the Global Offset Table (GOT) pointer for Position Independent Code (PIC) code are stored in the stack. A basic rule of a thread re-scheduling is that the core must retrieve the same state that it has, before the context switch procedure.

Algorithm 7 Thread Yielding (this is a thread safe function).

1: Comes from the user code
2: Broadcast notify to all PEs /* *Avoid deadlock when one thread runs on this core* */
3: Get the Stack pointer in the core context
4: Decrement the Stack pointer by the size of the register file
5: Write core registers' content at the Stack pointer address (save context)
6: Call the cooperative scheduler
7: Get the return value of the scheduler which is the current thread's Stack Pointer
8: Read the new core context from memory at the Stack pointer (restore context)
9: Increment of the Stack pointer by the size of the register file
10: Goes to the user code

We benchmarked the low-level primitive for yielding a thread in a CC of MPPA® at 204 machine cycles. All *Load/Store* operations have been programmed as uncached to pre-fetch data in advance and eliminate L1 misses during the context switch and for accessing the shared variables within the scheduler.

Thread Exit or Return

The thread exit or return allows the programmer to terminate the running thread. This operation changes the thread to the *Finished* state and calls the yield operation.

The NMTR commits all pending writes in the memory and updates the user-space thread object to *Finished* state. Then the scheduler, explained below in Section 6.4, cleans and removes the thread from the internal slots (compare-and-swap).

When the scheduler is called, the scheduler sets the state of thread object structure to *Exiting* state. The *Exiting* state of a thread makes the join operation possible of the thread object structure. The join operation is explained below.

The exit function is either explicitly called by the calling user thread or implicitly called when the user thread function returns. Once the thread becomes joinable, no interactions between the scheduler and the joinable thread are visible. Indeed, the user-space opaque thread structure contains all the required information for the join operation.

Thread Join

The join function allows any threads that own the thread structure address to join the exited thread. The join operation has a synchronization effect with the joined thread, as well as memory consistency maintenance between the joined threads and the thread initiating the joining operation.

Algorithm 8 Thread Join (this is a thread safe function).

```
1: Input: Thread_Object { Core_ID, Thread_ID, Core_State,
   Static_Memory_Slot_ID }
2: Core_State = load-uncached(&Thread_Object.Core_State)
3: Memory_Slot_ID = load-uncached(&Thread_Object.Static_Memory_Slot_ID)
4: slot_object_alloc_mask = load-uncached(&64bit_slot_obj_alloc_addr)
5: while Core_State is not Joined do
6:   if Core_State is Exiting then
7:     while true do
8:       if compare-and-swap(&Thread_Object.Core_State,
         Core_State, Joined) then
9:         if Memory_Slot_ID is dynamic then
10:          Free Memory_Slot /* Always success and atomic */
11:          goto exit_success
12:        end if
13:        while true do
14:          new_slot_object_alloc_mask =
             slot_object_alloc_mask | (1ULL<<Memory_Slot_ID)
15:          if compare-and-swap(&64bit_slot_obj_alloc_addr,
             slot_object_alloc_mask, new_slot_object_alloc_mask) then
16:            goto exit_success
17:          end if
18:          slot_object_alloc_mask = load-uncached(&64bit_slot_obj_alloc_addr)
19:        end while
20:      else
21:        goto exit_success
22:      end if
23:    end while
24:  end if
25:  Call Yielding to release the core (See Algorithm 7) /* Thread is running */
26:  Core_State = load-uncached(&Thread_Object.Core_State)
27: end while
28: Label exit_success: Return success
```

6.3 Synchronization Primitives

Synchronizations are unavoidable in parallel computing to satisfy multi-core Read-After-Write (RAW) data dependencies. The efficiency of the synchronization primitives is one of the keys to enable competitive multi-threading at fine-grained. Algorithm 9 shows the architecture of the synchronization primitives. These primitives are usable at low-level by the NMTR or using the *pthread* API for the *barriers, mutexes*, and *semaphores*.

Algorithm 9 Synchronization Primitive Algorithm (this is a thread safe function).

```
1:  init: The memory updated atomically is already initialized and consistent
2:  Input: Opaque Object Address
3:  Write memory barrier /* Stalls the core until all write accesses are completed */
4:  while true do
5:     boolean = Execute in memory an atomic operation /* Lock-free for performance */
6:     if boolean is true then
7:        break
8:     else if is an asynchronous call then
9:        Return try again flag
10:    end if
11:    Call the yield operation /* the scheduler is called */
12: end while
13: Full memory barrier /* Stalls the core until all read/write accesses are completed */
14: Broadcast notify to all PEs /* force all core to re-evaluate conditions */
15: Return success
```

Firstly, the CPU commits in memory all pending outstanding writes. The core uses a write memory barrier to make it possible (see Line 3 in Algorithm 9).

Secondly, the synchronization function attempts to perform an atomic modification in the memory (see Line 5 in Algorithm 9), on failure, the CPU either yields to another software thread if any or return to the user the *try-again* information for later calls.

Finally, upon success, the CPU executes a broadcast notification to wake-up all potential PEs waiting to be unlocked by the memory modification performed with a lock-free atomic operation (see Line 14 in Algorithm 9).

As the k1-VLIW core lacks memory coherency support, the runtime invalidates by default the L1 data cache when a lock is taken, when a semaphore token is taken, and when a barrier is completed. Indeed, the lack of hardware L1 data cache coherency means that the software must make sure that future *Loads* (Reads in the local shared memory) see the modifications of the writes in memory of other (physical) PEs or DMAs.

6.4 Cooperative Scheduler

The cooperative scheduler makes it possible to run several logical threads on a single core. When a thread is created, it is placed in one of the internal thread slots. The hardware core can execute up to NB_SOFT threads in a cooperative multi-threading mode. Unlike preemptive scheduling, cooperative scheduling makes the multi-threading very efficient as there are no interrupts. Interrupt issues are explained above in Section 6.2.2.

Each time the thread creation Algorithm 6 creates a new thread, the thread is placed in an internal thread slot and marked as runnable for NMTR. When the PE calls the

scheduler presented in Algorithm 10, it will schedule one of the runnable threads whenever a specific condition is satisfied. These conditions are explained in Table 6.1.

When executing the scheduler code, the core operates on the stack of the calling thread of the scheduler (this is not preemptive scheduling). Currently, there are no stack overflow checks; however, as stacks are setup at boot time (known by the NMTR), it is possible to check their current state each time the core enters the scheduler. It could also be possible to protect the stack using the MMU feature of the core; but, it was not done due to the lack of time and work priorities.

Algorithm 10 Cooperative Scheduler Algorithm (this is a thread safe function)

```
1:  Input: an opaque Event, Thread_Object { Core_ID, Thread_ID, Core_State,
    Static_Memory_Slot_ID }
2:  if Thread_Object.Core_State is Finished then
3:    /* Thread Returned or Exited */
4:    while true do
5:      slot_mask = load-uncached(&64bit_slot_mask_addr)
6:      Thread_ID = count-trailing-zero(slot_mask)
7:      if compare-and-swap(&64bit_slot_mask_addr, slot_mask,
        slot_mask | (1ULL<<Thread_ID)) then
8:        break
9:      end if
10:   end while
11:   Write memory barrier /* Stalls the core until all write accesses are completed */
12:   Broadcast notify to all PEs
13:   Streaming-store to write Exiting State to Thread_Object.Core_State
14:   Broadcast notify to all PEs
15: end if
16: /* Cur_Slot_Thread in range [0 ... NB_SOFT - 1] */
17: Cur_Slot_Thread = Threads_ID / NB_CORES
18: Nb_Soft_Thread = NB_SOFT - 1 /* to support core idle */
19: while true do
20:   for i in (Cur_Slot_Thread+1) ... (Cur_Slot_Thread + Nb_Soft_Thread) do
21:     New_Thread_ID = (i % NB_SOFT)*NB_CORES+Thread_Object.Core_ID
22:     if New_Thread_ID is Runnable then
23:       if New_Thread_ID event is NULL then
24:         Return success and Return New_Thread_ID stack pointer
25:       else if Specific test event handler on Event is true then
26:         Return success and Return New_Thread_ID stack pointer
27:       end if
28:     end if
29:   end for
30:   Nb_Soft_Thread = NB_SOFT /* Will need to check all soft threads now */
31:   Core idle state /* Woken-up by the doorbell */
32: end while
```

Scheduler: Condition Invocation

NMTR operates using a cooperative scheduling policy. Therefore, the scheduler is called only at some specific points in NMTR. These points are listed and explained in Table

6.1. When the scheduler is called from a thread, the scheduler tries to run another thread assigned to this core, if any exists. The thread calling the scheduler is then re-scheduled, if the condition (see Table 6.1) that made it yielded before, becomes true.

Table 6.1 – *Scheduler Condition Call on Standard Primitives for Cooperative Multi-Threading*

Standard Primitive	Scheduler Condition Call
pthread_yield	On the primitive call as it release the CPU
pthread_mutex_lock	On the primitive call, if the lock is already taken
pthread_barrier_wait	On the primitive call, if the number of contributors to the barrier is not reached
sem_wait	On the primitive call, if the semaphore do not have any available tokens
pthread_cond_wait	On the primitive call, if the condition is not satisfied (blocked)
pthread_exit	On the primitive call, the thread is destroyed unconditionally
Thread handler returns	In C/C++ on the return statement, the thread is destroyed unconditionally

Another specific case for calling the scheduler is right after the boot and initialization of our new NMTR. The scheduler is called on all cores not running the main() function. Then, if a new thread is created by the programmer, the selected idling core will immediately take this new thread and schedule it.

Scheduler: Condition of Thread Scheduling

The scheduler is local to each PE. Each thread is pinned to a unique PE. The scheduler executes threads in Round Robin (RR) policy. It executes the pinned threads in run-to-completion mode if any. The thread is descheduled only when it encounters one of the primitives explained in Table 6.1. When no threads are runnable for a given core, the core goes in the idle state. To avoid deadlocks (non-progress of any resources), it is important that the core checks all states of threads assigned to this core, before switching to the idle state.

Our newly NMTR makes it possible to associate an event to a specific condition (see the input of Algorithm 10). If this event is NULL (empty) and the selected thread is runnable, the thread is elected. If the event is not NULL and the selected thread is runnable, then a specific condition is evaluated. This mechanism enables conditioning of the schedulability of a thread on a predefined event occurrence. Indeed we use this feature for enabling thread activation on DMA transfer completions. Custom events can be built with user-defined conditions in memory or with other external events.

6.5 Using NMTR to Enable OpenMP Multi-Threading

Our new NMTR makes it possible to run an OpenMP multi-threading runtime, thanks to the provided set of features of our contribution. The needed features are the support of thread creations, the semaphores, and the mutexes.

6.5.1 Configuration & Architecture

The libgomp module of the GNU Compiler Collection (GCC) project has been available since 2006. In Section 6.1, our *newlib* library is a dependency when building the GCC compiler for generating code for the k1-VLIW processor. As presented in Section 2.3.2, the Compute Cluster (CC) is composed of 16 user PEs, and the Input/Output Subsystem (IO) is composed of 4 RMs with a shared L1 data cache. The OpenMP libgomp runtime back-end of GCC uses the *sysconf* POSIX standard primitive with the *_SC_NPROCESSORS_ONLN* argument to request the runtime for the number of available threads. By default, we return NB_CORES in both the Input/Output Subsystem (IO) and Compute Cluster (CC) of the MPPA® processor, respectively 4 and 16.

The definition of the Pthread API within the *newlib* library allows the GCC compiler to build the OpenMP runtime. Indeed we use the Pthread execution back-end, based on the Pthread API, to make OpenMP multi-threading possible.

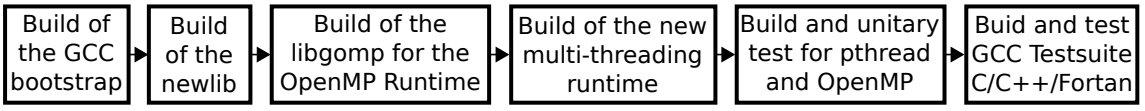


Figure 6.2 – *Build and Test Process for the Integration of the New Multi-threading Runtime in the Software Toolchain*

In Figure 6.2, the process for building and testing the new multi-threading runtime is presented. First, the GCC is built for generating code for the k1-VLIW core. Then, the *newlib* and our new multi-threading runtime are compiled. Finally, the OpenMP runtime is built and validated. All of these steps are run automatically each time a developer contributes to the project, integrating all of these tools and runtimes to avoid regressions.

6.5.2 Internal Contributions to GCC libgomp

The OpenMP runtime back-end of GCC, based on the POSIX threads, creates threads using the *pthread_create* primitive. For performance issues, the OpenMP runtime neither joins nor destroys the created threads. As such, the libgomp places the thread in a wait mode (sem_wait) when a parallel region is ended. On the Linux system, this is not a problem as the memory is usually at least a gigabyte, which means that hundreds of threads can be run. Also, the Linux system can clean up the threads and their used memory when the process is exiting (either on failure or success).

The NMTR does not implement the threads and memory clean up. If the threads started by the OpenMP runtime need to be recycled to do other things, they have to be joined and the memory they used must be freed in order to be recycled for later use. Freeing the memory let the user run legacy OpenMP code at some point, and starts new Pthread threads without using any additional memory. For OpenMP performance, the threads can be left in idle mode, ready to compute the next parallel region.

Destruction of the Parallel Region Threads

The standard OpenMP libgomp runtime of GCC has been modified to make the joining of threads possible. The OpenMP runtime already implements an internal primitive called the `gomp_free_thread` that makes the master of the OpenMP Team (master of the OpenMP parallel region) send exit messages to the threads of the parallel region. Each master thread stores information about the number of threads that were created for the parallel region in the TLS (See Section 2.4.2). In this TLS memory section, we save the `pthread_t` opaque pointers that reference the created threads by the OpenMP runtime. On `gomp_free_thread` call, we then iterate on the saved referenced threads and call the `pthread_join` primitive on these threads, to clean up all the used memory areas.

Thanks to this contribution, we can now mix OpenMP and `pthread` multi-threading without any extra memory usage such as the internal stack of threads or thread reference structures. In local memory-based processors, the on-chip memory is scarce and needs to be efficiently managed to reduce the main memory bandwidth, which is one of the main bottlenecks on such chips.

Optimization of the OpenMP Runtime

Another contribution to this runtime for efficient OpenMP is the implementation of a static pool of POSIX synchronization objects that are intensively used by the OpenMP libgomp runtime, each time the master thread encounters an OpenMP parallel region. We implemented a pool of statically allocated mutexes and semaphores, each composed of 32 slots. The allocation and freeing of these resources is performed using a lock-free mechanism based on a 32-bit variable updated atomically by `compare-and-swap` instructions in memory as in previous algorithms. Each bit set to 1 represents a free slot. When no static slots are available, the runtime automatically switches to the dynamic allocator provided by the *newlib* library, which is thread-safe.

6.6 Auto-threading: Automatic Thread Scheduling on RDMA Completion

Almost all applications optimized for the MPPA® processor and other DMA-enabled architectures try to overlap communications and computations. To do so, N-buffering techniques are used [ZK06], but they are complicated to implement and validate in real-life applications. Indeed, the explicit writing of software asynchronous DMA transfers by hand is tedious and error-prone.

The contribution presented in this section aims to automate the classical double or N-buffering technique of DMA-enabled architectures. The N-buffering technique makes the overlapping of data communications and computations possible. This contribution is inspired by the hyper-threading technology of CPUs (task-parallelism) or hardware multi-thread of Graphics Processing Units (GPUs) (data-parallelism).

Our contribution instead operates at coarser granularity, as the yielding and re-scheduling operations of a thread is within the hundred of machine cycles whereas, for traditional CPUs and GPUs, it is within the machine cycle. A practical consequence is that memory transfers must be big enough to cover to software overhead of context switching. To understand this section, it is advised to read Chapter 5. The contribution presented in this section is based on the Asynchronous One-Sided (AOS) API and runtime, that are presented in Chapter 5.

6.6.1 Auto-threading: Design and Implementation

The auto-threading mechanism requires support in both our new NMTR and Remote Direct Memory Access (RDMA) communication runtime. As such, the two runtimes need to be compatible with each other. As seen in Section 5.3.3 (AOS library in Chapter 5) in Algorithm 3, it is possible to yield instead of idling the core when the AOS event completion operation is not completed. Firstly, the weak dependency between the two runtimes is explained. Secondly, a typical use-case implementing the automatic yield onto RDMA is presented.

Enabling the Feature with Weak Dependency

Weak dependencies let the developer add objects in an Executable and Linkable Format (ELF) statically and efficiently. In a compiled object file, a 'weak' dependency symbol [3] is resolved at compile time statically if the symbol exists within the list of object files given to the linker. If not, the symbol is set to NULL statically. It makes it possible to add statically linked functions to object file very efficiently (the function pointer becomes immediate in the final ELF), and it is configured using the following link flag *-Wl,–defsym=foo=toto* where 'foo' will take the 'toto' address (case of the GNU Compiler Collection (GCC) linker). With such mechanisms, it is possible to either activate or deactivate at link time some features that are in the hot text path (*.text*).

A 'weak' dependency function has been implemented in our NMTR that can be overloaded at link time of the ELF.

Interaction with the Asynchronous One-sided Communication Engine

As this 'weak' dependency function is in the multi-threading runtime, the call to the test event function of the asynchronous one-sided API is now possible whenever the user wants to enable it. The call to the function can be seen in Algorithm 10 at Line 25. This function is called to test if a thread shall be re-scheduled or not. The thread becomes schedulable when the data requested by the Asynchronous One-Sided (AOS) communication library is returned and consistent in the memory. If not, the scheduler continues to test events and goes into the idle state when all potential NB_SOFT events are tested unsuccessfully. Each time an event pops in memory or the DMA Network on Chip (NoC) interface, the PE goes out of the idle state of the scheduler, and the PE tries again to test all potential NB_SOFT events.

Thanks this feature, it is possible to write N-buffering techniques automatically only by using more threads on a single core ($NB\_SOFT = 4$ threads at most). We provide a C example in Figure 6.3 to show how to use it in real-life application.

[3] https://gcc.gnu.org/onlinedocs/gcc-4.7.1/gcc/Function-Attributes.html

```
#define N_CORE (16) /* Compute Cluster has 16 cores */
#define N_BUFFERING (2) /* Number of buffer used */
#define LOCAL_SIZE (4096) /* local working set size */
#define LOCAL_WORKING_SET_NB (1024) /* number of working set to compute */

/* thread reference */
static pthread_t thread[N_BUFFERING*N_CORE - 1];
/* rdma segment */
static mppa_async_segment_t rdma_data_segment;
/* data in cluster's smem */
float local_data[N_CORE][N_BUFFERING][LOCAL_SIZE];

void* task(void *arg) {
  const int tid = syscall(SYS_gettid); /* thread ID */
  const size_t size = sizeof(local_data[tid][0]); /* Size to compute */

  for (int i = 0 ; i < LOCAL_WORKING_SET_NB ; i++) {
    if (mppa_async_get(
      local_data[tid][i%N_BUFFERING], /* local address */
      &rdma_data_segment, /* remote segment */
      size * LOCAL_WORKING_SET_NB * tid + size * i, /* offset */
      size, /* size of the linear data */
      NULL) != 0) { /* blocking but yields inside */
      assert(0 && "Failed get data\n");
    }
    /* yielding happen at the frontiers of kernel execution */
    kernel_compute(local_data[tid][i%N_BUFFERING]); /* compute */
    if (mppa_async_put(
      local_data[tid][i%N_BUFFERING], /* local address */
      &rdma_data_segment, /* remote segment */
      size * LOCAL_WORKING_SET_NB * tid + size * i, /* offset */
      size, /* size of the linear data */
      NULL) != 0) { /* blocking but yields inside */
      assert(0 && "Failed get data\n");
    }
  }
  return NULL;
}
int main(int argc, char *argv[]) {
  /* get reference on remote data in another memory (for rdma accesses) */
  if (mppa_async_segment_clone(&rdma_data_segment, 0/*segment id*/,
    NULL/*unused*/, 0/*unused*/, NULL/*blocking*/) != 0) {
      assert(0 && "Failed clone segment data\n");
  }
  for (int i = 0 ; i < N_BUFFERING*N_CORE - 1 ; i++) {
    if (pthread_create(&thread[i], NULL, task, NULL) != 0) {
      assert(0 && "Failed create thread\n");
    }
  }
  task(NULL); /* work of main thread */
  for (int i = 0 ; i < N_BUFFERING*N_CORE - 1 ; i++) {
    if (pthread_join(thread[i], NULL) != 0) {
      assert(0 && "Failed join thread\n");
    }
  }
  return 0;
}
```

Figure 6.3 – *Auto-thread onto RDMA Transfers for Automatic Double Buffering (parallel code)*

It is also possible to combine the automatic yielding onto RDMA transfers with OpenMP parallel regions. The purpose is to have several (at least 2) OpenMP teams (parallel regions) that are collectively re-scheduled to work together onto the data coming from the DMA. By doing this, the programmer easily implements communication and computation overlapping without coding complex explicit asynchronous RDMA transfers. The completion of RDMA jobs, the allocation of outstanding buffers and the buffer rotation are more straightforward to be performed thanks to the runtime. Indeed when attempting to hide the memory access latency, buffers usually have several states onto which they rotate, namely, the *read state*, the *compute state* and the *write state*, which are not trivial to manage in the application code.

6.7 Results, Comparisons and Discussions

6.7.1 Benchmarks

Several benchmarks are used to compare the original multi-thread runtime versus our NMTR. The execution times are measured on the MPPA® chip, on a single Compute Cluster (CC). We vary the number of Processing Elements (PEs) dynamically during the execution of the benchmark. The results are cross-checked with a sequential reference implementation at the end of the execution. The only change is the multi-thread runtime linked to the benchmark.

Elementary Primitives: Original Runtime Vs NMTR

When 16 PEs are running, the experimental results show that our lock-free NMTR outperforms classical lock-based implementation by a factor of 10 on the barrier synchronization primitives (collective operation). A factor of 15 is shown for the thread create operations, and 22 for the thread join operations.

The barrier is benchmarked by iterating over it ten thousand times in each started threads. No computation is performed between successive calls to the barrier in threads. In Figure 6.4, the overhead latency of the barrier corresponds to one call. A full memory barrier is issued each time a thread enters and leaves the barrier.

For the mutex and semaphore benchmarks, we use both synchronization objects as a locking mechanism to increment a variable in memory. In both multi-threading runtimes, a full memory barrier is performed when entering the synchronization primitive and when leaving it. For both the semaphore and the mutex synchronization mechanisms the new multi-threading runtime is better by a factor of 2. As the locked code section only consists of loading, incrementing, and storing the variable in the shared memory, the latency is mainly induced by the multi-threading runtime itself.

It can be observed in Figure 6.4 that the semaphores and the mutexes of NMTR have different latency, whereas they almost provide the same service. Indeed, a binary semaphore is equivalent to a mutex for the user. The performance difference is due to the implementation of the atomic in memory that is retry-free for the mutex and lock-free for the semaphore. The lock-free implementation of the semaphore uses a compare-and-swap with a loop because it requires a saturation to 0 when no tokens are available. Lock-free guarantees forward progress for one initiator, but the other one needs to retry on failure. The retry-free means that the atomic always succeed; therefore, the throughput is better, providing less overhead at the end.

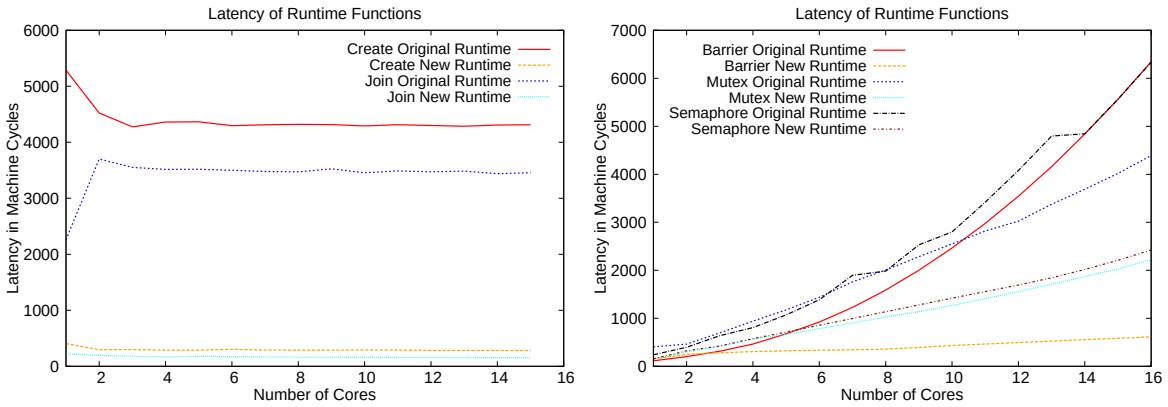


Figure 6.4 – *Performance Comparisons of Thread Creation, Join and Basic Synchronization Primitives on 16 Cores*

Elementary Primitives: What happens when more than 16 threads are used?

The original runtime does not support more than 16 threads, meaning that only one thread per core is possible. The new runtime of our contribution makes it possible to run up to NB_SOFT per core in both the CC and IO, as seen in previous sections. We run the same benchmark, but more threads are used ($\times NB\_SOFT$) in Figure 6.5. Results with less overhead are observed for the creation and join operations. Figure 6.5 also shows a linear increase of the latency for the mutex and semaphore, but the barrier remains very efficient. Again, the performance is due to efficient the retry-free atomics and the masked interrupts used as events to load and check conditional variables in the memory.

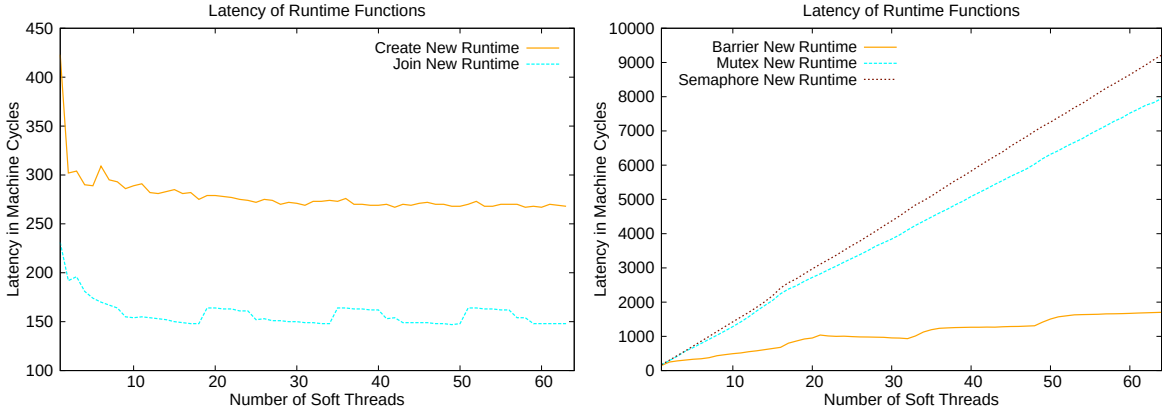


Figure 6.5 – *Performance Comparisons of Thread Creation, Join and Basic Synchronization Primitives on 64 Threads*

OpenMP Benchmarks

The original OpenMP runtime starts threads only once and leaves them internally created, waiting on a semaphore, ready to execute jobs for performance. As such, in the benchmark of this section, OpenMP teams of 16 Processing Elements (PEs) are already created, and we measure the speedup obtained onto *parallel for* regions.

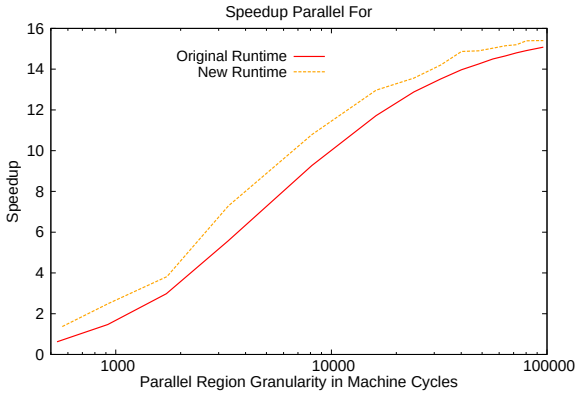


Figure 6.6 – *Performance of the OpenMP GCC libgomp, Based on our New Multi-threading Runtime with 16 Threads Running*

In abscissa, we vary the execution time (in machine cycles) of one parallel task of the parallel region. This measure of latency of the parallel task is done outside of the OpenMP region, sequentially, for fair measurements. Hence, the entire overhead of the OpenMP runtime is observed.

The two runtimes (the original one and our new one) are part of the GCC libgomp, but the new runtime implements static memory allocation for both `pthread` objects and some data structures of the GCC libgomp. Indeed, when the compiler uses our NMTR, as we modified the generic source of OpenMP runtime library of GCC for these optimizations, we automatically benefit from them.

As a result, our new runtime behaves more efficiently than the original one. In Figure 6.6, a gain of more than 10% is shown on fine-grained multi-threading using the OpenMP runtime based on our new multi-threading runtime. The gain is **automatic without modifying any source code line**. Also, our NMTR makes it possible to use simultaneously `pthread` and OpenMP multi-threading whereas the original did not.

Toward Higher Performance in the OpenMP Runtime for Fine-Grained Multi-Threading

In the future, the implementation of static pools of internal OpenMP data structures to bypass as much as possible calls to the dynamic memory allocator of the *newlib* will be considered. Another optimization is to re-implement the generic synchronization code to make it lock-free. The GCC OpenMP libgomp already implements such mechanisms (at some point), using standard builtins as explained in Section 4.5.4. However, some work is still required to make sure that this would be functional and more efficient than the current implementation. Finally, streaming loads for sharing data could also be implemented directly in the libgomp runtime. Such optimization requires much more effort as loads will need to be either explicitly written by hand or by using a named address space[4] as an attribute to pointers to make the compiler generate streaming load on memory accesses through this pointer. Also, it is often required to modify the code so that the loads are schedule earlier as the streaming-load latency for Read-After-Write (RAW) dependency is higher.

[4]https://gcc.gnu.org/onlinedocs/gcc/Named-Address-Spaces.html

Auto-threading

The presented results show the benefice of auto-threading onto RDMA transfers explained in Section 6.6. The data transfers are coded synchronously and are easily readable. We run three benchmarks, namely the copy, vector add and an image filtering operation. These applications are greedy in terms of memory. They show how efficient our new auto-threading mechanism is when yielding onto the completion of RDMA transfers. We measure the main memory bandwidth (external) of the application, that is linear with the execution time. The memory bandwidth of the application is defined as the throughput of the application. Without auto-threading, such synchronous data transfers induce stalls of the thread that initiates the data transfer(s); thus, it has less performance as we see in Table 6.2. Such measures provide the efficiency of the memory throughput which is the purpose of the auto-threading feature, that is, hide the memory access latency.

1 Cluster - 1 PE Benchmark Size data set	**Auto-threading Disable GB/s**	**Auto-threading 2 Thread GB/s**	**Auto-threading 4 Threads GB/s**
copy 4 Mega Bytes	2.8	3.03	3.04
vector add 4 Mega Words	1.0	1.59	1.6
Filter 1x1 - 1080p 4 x 8-bit channels	0.348	0.404	0.400

Table 6.2 – *Auto-threading Throughput on Three Different Use-cases*

On the vector add benchmark, we show an execution time speedup of 60% when enabling the auto-threading feature. The stall time of the PE onto the waiting of the completion of RDMA transfers is almost reduced to the overhead of a context switch. In this case, the system is not bound by the memory wall as only one PE is used. Performance could be better if we optimized the benchmarked use-case in assembly using classical optimization like Single Instruction, Multiple Data (SIMD) and streaming memory access with packing. The kernel could also be parallelized in the CC, using only one PE to perform the RDMA communications.

6.8 Conclusion

This chapter proposes a new implementation to make lightweight multi-threading possible on a multi-core Symmetric Multi-Processor system (SMP). Main problems are explained, and the low-level implementation is provided to overcome the issue of standard locking mechanisms [MP92]. Performance improvements here are mainly due to the lack of optimization in the original runtime. Indeed, the original runtime uses locking mechanisms when atomicity and commit orders are required to accesses a shared resource. We instead use lock-free mechanisms for atomic updates of shared data structures and order the memory accesses to them. This chapter shows that writing such runtime is not a trivial task as it requires mastering the architecture along with managing the memory consistency and coherency of low-level parallel software at the system level.

OpenCL Task-Parallel

This new multi-threading runtime is used in the proof-of-concept of the OpenCL task parallel runtime, but also in the OpenCL data parallel runtime running in production. The OpenCL task parallel model was elaborated in collaboration with Minh Quan during his Ph.D. thesis. In OpenCL data parallel model, each PE of a Compute Cluster (CC) (Compute Unit), is a Work-Group as explained in Section 3.2.2. Such a strategy represents a severe bottleneck regarding the local (`__local`) memory of the Compute-Unit as it cannot be shared across cores of the same Compute Cluster (CC) (Compute Unit) (64 KB of local memory maximum). The OpenCL task parallel model maps a Work-Group on a single Compute Cluster (CC) meaning that only one PE executes the Work-Group and the threads need to be managed inside the Work-Group. Such a mode allows much bigger local memory size (1 MB), that is useful for advanced data pre-fetching and for keeping the hot data path as close as possible to the PEs within the Compute Cluster (CC). With assembly optimization, larger local memory for asynchronous RDMA prefetching and explicit multi-threading inside the CC, Quan and I obtained an execution time three times better than the original OpenCL data parallel, using the OpenCL task parallel mode for a General Matrix Multiply (GEMM) $4096 * 4096$ use-case [KVL91].

Maturity, Standard Tests, and Today's Usages

Our NMTR has high maturity. NMTR supports the most used `pthread` primitives like mutex, spin, barrier, and semaphores. It also supports the TLS data section that is heavily used in the *newlib* and the OpenMP runtime of GCC. The runtime also supports the entire C/C++/Fortran OpenMP standard test suite of GCC 4.9. Finally, since late 2017, it is used in most Kalray's products internally, and it is today the default multi-threading runtime since AccessCore 3.0 (Kalray's toolchain name). For instance, it is used in the OpenCL runtime, Kalray Neural Network (KANN) runtime, the OpenVX framework presented in Chapter 9, the Synchronous Parameterized Interfaced Dataflow Embedded Runtime (SPIDER) runtime presented in Chapter 8, and the Kalray's networking solutions.

CHAPTER 7

Software Synthesis based on a Hierarchical Static Dataflow Model for a Clustered Manycore Processor

This chapter presents a new strategy for mapping a static dataflow programming model the targeted manycore. Indeed, manycore processors are not widely deployed due to their programming complexity, and because applications are not adapted to these architectures. To exploit the performance of complex clustered manycores, the application has to be split and mapped onto the available cores. This task is complex and time consuming. Dataflow programming models inherently make it possible as they represent an application with a set of actors (functions) communicating between each other with First-In-First-Out queues (FIFOs) (data) as already seen in Section 3.3.

Computer system architectures are more and more complex. They often implement more Processing Elements (PEs) and memories. Computer system architectures are hierarchical in terms of memory architectures (caches and local memories) but now also concerning Processing Elements (PEs) architecture (i.e., clusters of cores for instance). It is a way to both increase the computation capabilities and keeps the system under control.

Research on dataflow modeling leads to the continuing introduction of new dataflow models. The hierarchy has been introduced in several dataflow semantics. For instance, static extensions of the Synchronous Dataflow (SDF) model such as the Interface-Based SDF (IBSDF) [PBR09] and the Compositional Temporal Analysis (CTA) models have been proposed to enforce the compositionality of applications. The hierarchy of these models enhances the expressiveness and conciseness of the models while preserving their predictability. A model is compositional if the properties (schedulability, deadlock freeness) of an application graph composed of several sub-graphs are independent of the internal specifications of these sub-graphs [Ost95]. IBSDF interfaces are inherited by the Parameterized and Interfaced dataflow Meta-Model (PiMM) meta-model and its application to the SDF programming model called Parameterized and Interfaced SDF (PiSDF) [DPN+13].

Previous works on Parallel and Real-time Embedded Executives Scheduling Method (PREESM) use the hierarchy feature to ease the application description. IBSDF has first proved to be an efficient way to model dataflow applications [PAPN12], and most of the applications developed using PREESM, and the PiSDF programming model uses the hierarchy feature. More recently, Deroui and al. used the hierarchy feature of IBSDF for the fast throughput evaluation of applications [DDNMK17a, DDNMK17b].

The hierarchy feature is not used so far in the mapping/scheduling and the code generation of PREESM. Hierarchical graphs are flattened before they are processed. Flattening all the hierarchy is problematic to process large dataflow graphs for architectures with hundreds of cores. As seen above, the mapping and scheduling problems are known to be NP-complete. The time to compute the scheduling and the mapping increases exponentially with the number of actors to map and with the number of PEs of the targeted computer. The memory allocation is also an NP-hard problem as already mentioned in Section 3.3.4. The use of flattened graphs may also increase the number of synchronizations between PEs during the execution and thus actively degrade the overall system performances. When Piat and al. defined the IBSDF programming model [PBR09], the main idea was to use the hierarchy levels as code closures. The IBSDF fosters sub-graph composition making sub-graph executions equivalent to imperative language function calls. This idea has not been used so far in the code generators of PREESM.

In this chapter, we show that the hierarchy of the dataflow graphs can be used to program efficiently hierarchical computer system architectures. To do so, we exploit the dataflow graph hierarchy of the Interface-Based SDF (IBSDF) model. The upper hierarchy levels of the IBSDF graph are used for the mapping/scheduling between clusters of the Multi-Purpose Processor Array (MPPA), called the coarse grain mapping/scheduling. The hierarchical approach thus reduces data movements between clusters and increases the arithmetic intensity inside clusters. The arithmetic intensity is the amount of processing done for each byte of data transferred to a Compute Cluster (CC). Maximizing the arithmetic intensity is essential to achieve decent performance for applications running on a clustered architecture like MPPA® or any other multi-core Central Processing Units (CPUs).

The lower levels of the graph are used for the mapping/scheduling inside the clusters. It is called the fine-grained mapping/scheduling. Repetitions of an actor in the IBSDF graph are analyzed and clustered to generate code including Open Multi-Processing (OpenMP) primitives and for loops. The MPPA® toolchain compiles this code, and it is executed in parallel inside one cluster. This approach makes the mapping and the scheduling of static dataflow application graphs onto the PEs of a clustered manycore architecture faster, while preserving the parallelism of the application.

The contribution presented in this chapter has been designed, implemented and tested in the open source project PREESM. It has been integrated since the PREESM release 2.3, and it requires the Kalray AccessCore release 2.9 or higher for compilation and execution.

This chapter is organized as follow. First, our strategy for targeting manycore processors using IBSDF graphs is presented in Section 7.1. We detail how the coarse grain and the fine-grained parallelisms of the applications are efficiently exploited in Section 7.2. Then, Section 7.3 presents the graph *Clustering* transformation. Finally, Section 7.4 presents benchmark results, discusses limitations and provides hints for enhancements.

7.1 Hierarchy of IBSDF to Target a Hierarchical Manycore Processor

This section explains how the mapping and scheduling can benefit from the hierarchical feature in IBSDF. We illustrate this contribution with an image processing application. This application and its IBSDF hierarchical dataflow graph are presented in Section 7.1.1. Section 7.1.2 explains the design, the choices, and the implementation in the PREESM tool.

7.1.1 A Hierarchical Dataflow Application

We consider an image processing application described by its IBSDF graph in Figure 7.1. The purpose of this application is to apply a commonly used *Sobel* image filter and two morphological operators (one *Erosion* and one *Dilation*) to detect the edges of the processed image.

The IBSDF graph is composed of six actors at the top level of the hierarchy. Three of them, with red borders, are hierarchical actors. Each of the three sub-graph includes a single actor. The actor production and consumption rates are given by a parameter m. This parameter gives the number of execution of the actors. Indivisible data tokens exchanged in this graph are pixel lines of width w.

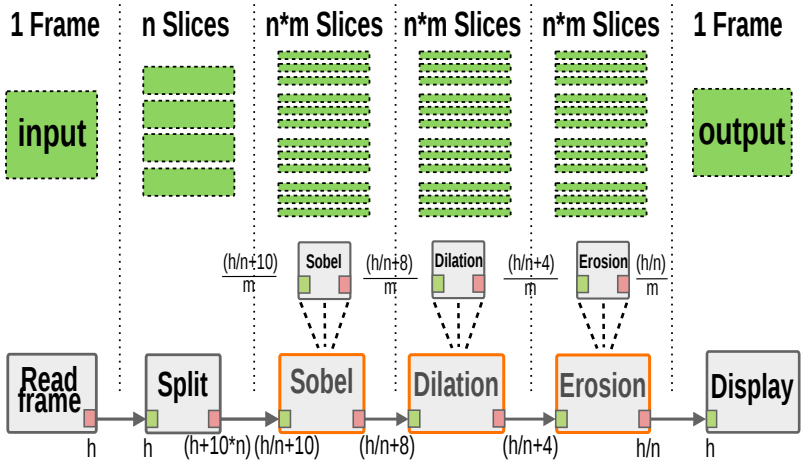


Figure 7.1 – *IBSDF Graph: Edges Detection and Denoising*

Using the original PREESM workflow, introduced in Section 3.5.1 and represented in Figure 3.12, the *Hierarchical Flattening* operation replaces hierarchical actors with their sub-graphs.

In our use-case, when the *Hierarchical Flattening* is performed, the number of automatically generated actors after the *Single-Rate* transformation becomes $3 * n * m$. The output of the *Hierarchical Flattening* transformation is the input of the *single-rate* transformation. The *Single-Rate* transformation reveals the whole parallelism of the dataflow application. In our case, the Repetition Vector (RV) is used to extract the data-parallelism of the sub-graphs. The RV is explained in Section 3.3.4. The actors generated by the *Single-Rate* and the *Hierarchical Flattening* transformations have to be mapped on PEs. Precisely, $3 * (n * m + 1)$ actors and $(4 * n * m + 1)$ communication edges are generated which have to be handled by the scheduler and mapper. Many more would be for complex application graphs.

7.1.2 Strategy: A Trade-off between Levels of Hierarchy

Our new hierarchical approach consists in exploiting the graph hierarchy in the different steps of the development flow instead of systematically flattening it. In the proposed method, we propose to specify whether or not the hierarchical actors should be flattened. In Figure 7.2, the *Hierarchical Flattening* operation is executed with a certain depth, meaning that the workflow will not flatten all the graph but only up to the specified depth. In the case of the graph presented in Figure 7.1, the depth of the flattening transformation is zero. Therefore, the graph is left unchanged for all actors in the graph. The exploitation of

the different levels of parallelism is presented in Section 7.2, and the clustering is presented in Section 7.3. As a summary, the method consists in exploiting several granularities of parallelism captured by nested, non-flattened, hierarchical graphs.

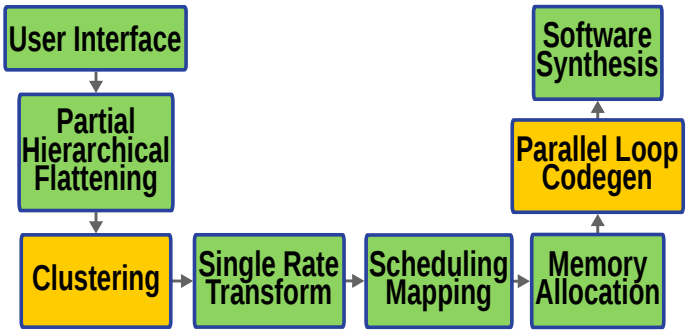


Figure 7.2 – *New PREESM Workflow for Clustering and Parallel Loop Generation*

We define a *clustered actor* as a non-flattened hierarchical actor. Clustered actors have larger memory footprints and execution time. They can be mapped onto one core as a single actor. Thus, the complexity to map a single actor is simpler than mapping the equivalent set of actors resulting from a flattening of the graph and its associated single-rate.

7.2 Exploiting Efficiently Two Levels of Parallelism

In the targeted architecture, two levels of parallelism are exploited: the coarse-grained and fine-grained parallelisms. Coarse-grained parallelism is found at the top-level of the hierarchy, where the graph contains clustered actors. Fine-grained parallelism is retrieved in sub-graphs of non-flattened hierarchical actors. In our case, the fine-grained parallelism is extracted from RVs during the software synthesis of hierarchical actors.

This approach aims to simplify the scheduling and mapping for clustered manycore processors but also off-the-shelf Symmetric Multi-Processor system (SMP) processors and multi-core Digital Signal Processor (DSP). Indeed, the software synthesis for hierarchical actors produces *For-Loops*, for loops like in the C semantics, that are exploited by compiler optimizations to extract the Instruction-Level Parallelism (ILP) (i.e., unroll and jam), or by OpenMP compilation passes that generate parallel code. Therefore, our proposal takes advantage of both high-level parallelism presented in Section 7.2.1 for the Compute Cluster (CC) (but also for the PEs for SMP architectures) using the hierarchical mapping strategy, and low-level parallelization explained in Section 7.2.2 for PE level. Section 7.2.3 explains how the software synthesis of explicit communications is performed. Optimization choices are provided, as well as a comparison to explicit handwritten communications using standard communication libraries.

Our approach lets the programmer select the most adapted hierarchy level. The PREESM programmer may decide what must remain hierarchical and what should be flattened. The scheduling is computed after the *Single-Rate* operation is applied.

7.2.1 High-Level Hierarchy (Inter-Cluster)

The mapping of the high-level hierarchical actors is used for coarse-grained parallelism granularity. The high-level is applied to inter-cluster parallelism. The coarse-grained mapping is supported by the hierarchy feature of the used dataflow model and has several advantages.

Firstly, the hierarchical actor software synthesis makes it possible to generate memory access automatically coalescing for actors of their sub-graph, whereas flattening the hierarchy generates many smaller data transfers (one for each firing of the actor in the flattened sub-graph). Memory access coalescing is a key to increasing the performance of the memory system. The sizes of the memory transactions are bigger; therefore, fewer memory transactions are required for a given use-case.

Secondly, the hierarchical actor software synthesis reduces the mapping complexity drastically when the number of PEs and actors increases. The CCs of the MPPA® processor are seen each as a single multi-core CPU, when our new coarse-grained application mapping is used. On a clustered architecture, hierarchical actors are mapped on a CC, whereas actors of sub-graphs are mapped at the core level.

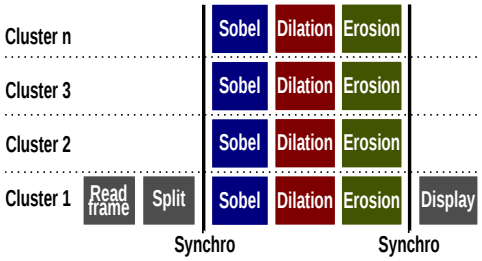


Figure 7.3 – *Gantt Chart of the Hierarchic Scheduling*

Figure 7.3 shows the inter-cluster parallelism at the top-level of hierarchy. The *Read frame* actor reads the images in a file stored on a hard drive disk, using stubbed system calls (open, read, and close). The CCs run concurrently in parallel the mapped hierarchical actors, *Sobel*, *Dilation*, *Erosion* hierarchical actors, and perform inter-cluster synchronizations. The *Split* does not copy any data but makes it possible to split the work of the CCs from 1 to n. Then the data are copied in the local memory of the CCs and computed. Once the parallel region of the CC is ended all contributors merge all results to the *Display* actor.

The proposed technique provides efficient usage of the on-chip memory and coalescing for data transfers. Moreover, the on-chip and off-chip memory are automatically allocated using the heterogeneous memory static allocator described in [DPNA16]. The used memory allocator supports both distributed memory and shared memory architectures.

7.2.2 Low-Level Hierarchy (Intra-Cluster)

The new support of hierarchical actors mapping and code generation allows both code factorization and more computational efficiency on fine-grained parallel code regions. The low level exploits intra-cluster parallelism. Fine-grained parallelism implies several concurrent computations, where the synchronization and memory consistency need to be managed efficiently. The generated code sections of hierarchical actors are automatically parallelized using OpenMP (if available). The intra-cluster parallelism is automatically extracted from the Repetition Vectors (RVs) of actors that have a potential source of parallelism in sub-graphs.

For instance, if a hierarchical actor A consumes $N*M$ tokens and actor B in hierarchical actor A consumes N, an RV of M (see Lines 8, 12, 16 in Figure 7.4) is automatically extracted and printed by the parallel code generator. This new feature allows for the automatic extraction of the RVs in sub-graphs, that contains actors that are a potential source of parallelism. In most cases, the number of loop iterations is known, as the firing of

actors is known thanks to the Directed Acyclic Graph (DAG), but also as the SDF graph is schedulable. The loops are generated in C language using a static finite *For-Loop*. A *For-Loop* is sequential, but our new software synthesis adds an *"omp parallel for"* to get parallelism for any architecture supporting the multi-threading model of OpenMP 3.0.

The parallel section is automatically inserted unlike [CPG+]. In this case, OpenMP 3.0 is very efficient as the number of *For-Loop* iterations is known at compile time. Thus thread-level parallelism is used for the hierarchical actors that are mapped onto the CCs. Synchronization points are the fork and join of the OpenMP runtime (see Lines 7, 11, 15 in Figure 7.4). The memory consistency points and the placements of synchronization points are known at compile time. Therefore, at the cluster level, the execution time is predictable, not only for sequential execution but also for parallel execution when using OpenMP 3.0 on finite *For-Loops*.

7.2.3 Automatic Generation of Explicit Communications between Clusters

On clustered architectures, the efficiency of data communications is crucial. A significant contribution is the use of automatically generated **Remote Direct Memory Access (RDMA) explicit memory accesses** transfers.

We show that explicit Direct Memory Access (DMA) communications outperform the shared memory approach provided by data caches through *Load/Store*. Moreover, the programmer does not need to know what happens at neither compile-time nor runtime. Indeed, as the application is already broken into pieces, thanks to the dataflow model, the automatic generation of explicit data communications is entirely hidden from the user once the dataflow application is described. Coherency, consistency, and synchronizations are also dealt with automatically by our new PREESM tool.

Our code generation is similar to [HDB+12], based on OpenMP 3.0, mixed with MPI-3 and using one-sided communications. The main difference is that our parallel code is generated automatically from an IBSDF dataflow graph and not handwritten as in [HDB+12]. The one-sided communication engine, used in this chapter for targeting the MPPA® processor is presented in Chapter 5. The one-sided operations make it possible for the Compute Clusters (CCs) to access asynchronously or synchronously the main off-chip memory and other Compute Clusters (CCs) local memories.

Regarding MPPA®'s new code generation support, the Compute Clusters (CCs) are seen as a single multi-core CPU for mapping at the coarse-grain level. The generated code is very close to the OpenMP mixed with Message Passing Interface (MPI)-3 [HDT+15] [HDB+12] using one-sided communications. However, we are here able to automatically generate parallel code using an IBSDF dataflow graph as input.

Concerning inter-cluster parallelism, the synchronizations are performed by explicit transfers directly across CCs at lines 2 and 22 in Figure 7.4. With our current software synthesis, inter-cluster data transfers go to and from the external memory (Double Data Rates (DDRs)).

The Compute Clusters (CCs) perform explicit memory data transfers that are based on highly efficient RDMA *Put Get* memory accesses thanks to the local memories. In Figure 7.4, RDMA accesses are done with *put* and *get* primitives at lines 19 and 4 respectively. The *wait* primitives ensure the RDMA transaction completion at lines 5 and 20.

Such code generation pattern has several advantages for an architecture like the MPPA® but also for other DSPs or general purpose processors. On the MPPA®, we automatically perform coalesced memory accesses at code generation as shown in Figure 7.4.

```
1. /* Inter-Cluster Synchronization */
2. synchro(...);
3. /* Main to Local Memory Coalescing */
4. get(..., tag); /* reads buffer */
5. wait(tag); /* wait end of transfer */
6. /* Parallel Hierarchical Sobel */
7. #pragma omp parallel for /* intra-cluster */
8. for(int i=0;i<M;i++)
9.       sobel(...); /* Sobel Kernel */
10. /* Parallel Hierarchical Dilation */
11. #pragma omp parallel for /* intra-cluster */
12. for(int i=0;i<M;i++)
13.      dilation(...); /* Dilation Kernel */
14. /* Parallel Hierarchical Erosion */
15. #pragma omp parallel for /* intra-cluster */
16. for(int i=0;i<M;i++)
17.      erosion(...); /* Erosion Kernel */
18. /* Local to Main Memory Coalescing */
19. put(..., tag); /* send buffer */
20. wait(tag); /* wait end of transfer */
21. /* Inter-Cluster Synchronization */
22. synchro(...);
```

Figure 7.4 – *Generated Code Example inside the CCs of the Manycore Processor*

Memory coalescing means that multiple data transfers are merged in one. It allows both the reduction of main memory data requests (requests traffic) and optimizes the usage of the local memory (local memory). When chaining kernels locally (i.e., IBSDF actors), without any communications other than intra-cluster communications, and synchronizations (shared memory), the execution overhead is very small.

The automatic optimization provided by our code generated limits data movements that are both very time and power consuming. The code generated in Figure 7.4 illustrates what is done on dataflow applications when both spatial and temporal data locality is exploited.

7.3 Automatic Clustering of IBSDF Graph

This section presents the new *Clustering* operation shown in Figure 7.2, represents the workflow described in this chapter. We explain the design and implementation of the algorithm itself in Section 7.3.1. Section 7.3.2 contents the different heuristics regarding clustering decisions, the clustering rules and the modeling of internal loops. The modeling of internal loops is the intermediate representation that is later used to synthesize parallel loops.

7.3.1 Algorithm: Design and Implementation

The clustering of a dataflow graph groups two adjacent actors (nodes) of the graph. The clustering algorithm is an automation of the Pairwise Grouping of Adjacent Nodes (PGAN) theorized by S. Bhattacharyya in [BML12]. The *Clustering* operation has a side effect which is the loss of the inherent parallelism expressed in SDF models. When actors are grouped, a nested loop schedule is built as represented in Figure 7.5. The loop iterations are set according to the Repetition Vectors (RVs) of the grouped actors. Thanks to those nested loops, it is possible to make them run in parallel using data parallelism.

The pseudo-code presented in Algorithm 11 gives the automated process of the *Clustering* workflow. In practice, only acyclic graphs can be clustered. As shown in Figure 7.2, the *Clustering* is performed after the *Hierarchical Flattening* operation, where the level of flattening is specified by the programmer.

Algorithm 11 operates on the entire top level graph. The algorithm retrieves the remaining hierarchical actors off the top level graph, and, it gets each associated graph recursively to flatten everything.

Then the clustering is executed $N - 1$ times. N is the number of actors constituting the sub-graph to be clustered. At Line 9 of Algorithm 11, the 'Get two mergeable actors' process is an external function of the algorithm that follows specific rules and where different strategies can be tested, as explained in Section 7.3.2. Once the sub-graph is completely clustered, a schedule of nested loops is generated. These loops communicate with each other; therefore, memory buffers have to be set in the memory to make it possible. Currently, this static memory allocator uses a single buffer, and a pointer is incremented each time a loop iteration is started. However, more advanced static memory allocation techniques can be used as already presented and implemented in Desnos [Des14].

Hierarchical actors that are neither flattened nor clustered will generate errors in the synthesis workflow. Therefore, after the clustering operation, hierarchical actors are associated with nested loops. These nested loops, their related actor functions, their input buffers, and their output buffers are printed during the software synthesis.

Algorithm 11 Pseudo Code of the *Clustering* Algorithm

```
1:  Input: Top-Level Acyclic Graph: G(E,V)
2:  Output: Acyclic Graph
3:  List of HierarchicalActors = Build list of all Hierarchical Actors of G(E,V)
4:  for Actor in List of HierarchicalActors do
5:     Loops = Create loop
6:     SubGraph = Getting associated graph of Actor
7:     FlatSubGraph = Flattening of the SubGraph (until no more hierarchy)
8:     while FlatSubGraph not contain 1 Actor do
9:        TupleOfActors = Get two mergeable actor(FlatSubGraph)
10:       FlatSubGraph = Construct the new graph (FlatSubGraph)
11:       Add TupleOfActors in the Loops Model in right order
12:    end while
13:    Memory = 0 (Initialization of number of bytes consumed of Actor)
14:    for TupleActor in Loops do
15:       Memory += Allocate internal working memory for the TupleActor
16:    end for
17:    Add Loops to the attributes of Actor (for software synthesis)
18:    Add Memory to the attributes of Actor (for software synthesis)
19: end for
20: Return G(E,V)
```

7.3.2 Clustering Rules, Heuristics and Loop Modeling

This section explains what has been automated regarding the clustering decisions and rules. The *Clustering* workflow has been split into two main parts.

The first part is the algorithm explained in Section 7.3.1. It links the clustering process with the rest of the framework, and it calls the clustering decision method (Line 9 of

Algorithm 11). The second part is the clustering algorithm itself. It takes the clustering decisions.

Rules Grouping together two actors that are adjacent is not a sufficient constraint. The two actors must also meet the following constraint.

Let's consider, the grouping of actor A and B (AB in this order), where B depends on A. The union of the successors of A and the predecessors of B must be empty. More formally: $Predecessors(B) \cup Successors(A) = \varnothing$. When such a constraint is satisfied, the two actors can be merged. We consider rA and rB as the Repetition Vector (RV) of actor A and B respectively. As shown in [BML12], the formal and factorized forms for representing the two new clustered actors is represented as follow:

$$GCD(rA, rB)\left[\left(\frac{rA}{GCD(rA, rB)}\right)A\left(\frac{rB}{GCD(rA, rB)}\right)B\right]$$

Where the GCD function computes the greatest common divisor.

Loop Modeling The outcome of the clustering algorithm is a sequence of For-Loops. A For-Loop implements a scalar that represents the number of iterations and the function to be run iteratively. When considering the previous example that was grouping the A and B actors, we show here the generated loops in Figure 7.5. rA and rB are static meaning that the *gcd* variable is statically evaluated and computed by the clustering workflow. The *gcd* variable is the greatest common divisor of rA and rB.

```
for(int i = 0 ; i < gcd ; i++){
  for(int j = 0 ; j < rA / gcd ; j++){
    /* Let's give the proper offset in the buffer */
    call_A(ptr_a + i * rA / gcd * size_a + j * size_a, ...); /* A */
  }
  for(int k = 0 ; k < rB / gcd ; k++){
    /* Let's give the proper offset in the buffer */
    call_B(ptr_b + i * rB / gcd * size_b + k * size_b, ...); /* B */
  }
}
```

Figure 7.5 – *Generalized Nested Loop Generation*

The input and output pointers need to be set according to the loop iterations. Indeed the buffers, precisely the pointers ptr_a or ptr_b in Figure 7.5, are given to the generated low-level functions within the C code software synthesis back-end. Such loops are very similar to the OpenCL data-parallel mode, where the work-group uses the global id and local id to address the memory. However, OpenCL computes these ids dynamically whereas our new software synthesis does it statically. However, the input is a static Interface-Based SDF (IBSDF) graph and not an OpenCL code.

Heuristics On complex graphs, the efficiency of the clustering depends on the merging order of actors. As of today, we implemented a simple clustering method that takes randomly two nodes (actors) and makes sure they respect the defined rules explained previously. It is a big limitation that is due to work priorities of this thesis. However, it is possible to implement other heuristics that can bring better loop builds regarding the parallelism degree and memory usage. Some ideas that could be tested are listed below:

- Select Actors with the bigger Repetition Vector (RV) first
- Select Actors with the smaller Repetition Vector (RV) first
- Select Actors with the bigger memory footprint first for minimizing the communication (most promising [CSWZ16])
- Select Actors with the smaller memory footprint first

Such clustering parameters can then be added as a parameter to the workflow to be tested accordingly depending on the dataflow application graph.

7.4 Experimental Evaluation

Our example is an image filtering application consisting of basic image processing steps, namely the sobel, erosion, and dilation kernels. Benchmarks have been run with a VGA resolution ($640 * 480$) for all architectures. The main purpose of this experimental evaluation is to show that the proposed hierarchical code generation has benefits for both mapping/scheduling as well as for the memory allocation. All benchmarks have been compiled using the GNU Compiler Collection (GCC) using *-O3* optimization. No assembly nor intrinsic optimization are used, as the main goal is to show our ability to exploit automatically both the parallelism and the data locality of a dataflow application.

Kalray MPPA®: Regarding the benchmark environment, the MPPA® is plugged into the motherboard of an Intel host processor where MPPA®'s Input/Output Subsystems (IOs) perform Peripheral Component Interconnect Express (PCIE) communications at runtime. Two execution modes are used. The first one uses the software emulated L2 cache where main memory accesses are done by *Load-Store*. The second uses explicit RDMA operation to perform one-sided memory accesses as presented in Chapter 5. A code sample for one-sided operations is listed in Figure 7.4. We focus our analysis on explicit memory accesses over RDMA, as the software emulated L2 cache provides lower performances because of irregular memory access patterns.

Inside the Compute Cluster (CC), the Kalray's proprietary Operating System (OS) runs an OpenMP implementation based on GNU Compiler Collection (GCC) libgomp as seen in Chapter 6. When the L2 cache is not used, the buffer allocation is done by [DPNA16], the generated code size, the OS size, and the library sizes should never exceed the 2 megabytes of local memory for each CC. If it is the case, both the workflow and the runtime (for advanced users) outcome an error.

Texas Instruments (TI) C66X: Texas Instruments (TI) C66X runs 8 DSP cores at 1 GHz. This Multiprocessor System-on-Chip (MPSoC) has a hardware L2 data cache enabling accesses to the main memory. IO communications are managed before and after running the application. Paper [SJA+13] presents the efficient bare-metal implementation of OpenMP multi-threading for the TI C66x.

Intel® Core i7: The Intel® Core i7 is a high-end Sandybridge architecture operating at 3.6 GHz with a DDR3 technology as main memory. The OS is a Linux system, and the used OpenMP runtime is based on the GCC libgomp library.

7.4.1 Results and Comparisons

This section presents the strong scaling results for three multi-core architectures, but the main focus is given to the Kalray's manycore processor. Table 7.1 presents the measured performances using the hierarchical actor software synthesis presented in 7.2.2. Compared to the single-core execution, a fair speedup is achieved on the TI C66X, with a maximum speedup of 7.2 on 8 cores. The Intel®Sandybridge off-the-shelf processor also presents fair speedup, up to 4.2, which is fair for an architecture with 4 physical cores (hyper-threaded).

Strong scaling is defined as how the solution time varies with the number of processors for a fixed total problem size. Strong scaling is important as it shows how efficient is the parallel strategy of the application when the number of core increases. Moreover, it shows how efficient is the additional code or the sequential code that is used to control the parallelization of the application. If the multi-threading software runtime has indecent performance, the strong scaling might rapidly saturate and drop due to the overhead of the software thread scheduling. Also, bad strong scaling results can also be due to a fined granularity of the parallel region.

Multi-core CPUs	TI C6678 EVM 1 GHz		Core i7-3820 3.6 GHz	
Nb Cores	FPS	Speedup	FPS	Speedup
1	8.9	1.0	49.3	1.0
2	17.6	1.9	91.6	1.8
4	33.8	3.8	155.6	3.1
8	64.4	7.2	211.5	4.2

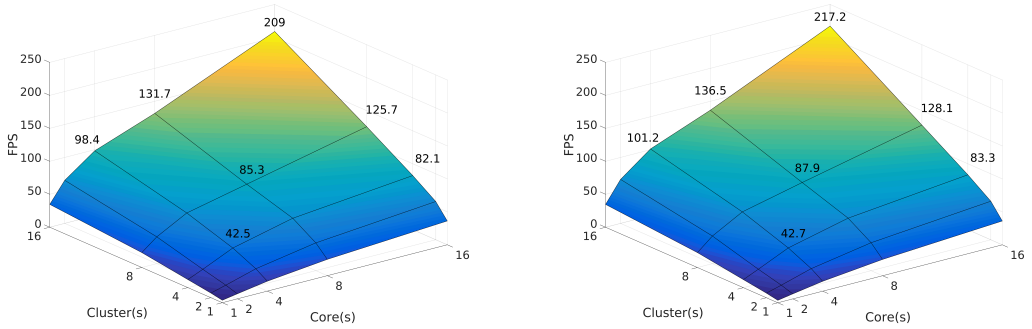
Table 7.1 – *Frames per second (fps) and Speedups for TI DSP and Intel Processor*

Table 7.2 shows mono-cluster (CPU of 16 Very Long Instruction Word (VLIW) cores) results using explicit communications and the distributed shared memory which emulates a software L2 data cache for off-chip memory accesses. As shown in Figure 7.4, the software synthesis that uses explicit memory accesses with RDMA outperforms the shared memory approach over the data cache up to **22%**. This table shows speedups of 13.4 when using explicit communications, and 11.2 when data accesses are performed by L2 data cache. Therefore, the scalability is efficient in both cases.

Mono-cluster MPPA®	MPPA® 400 MHz L2 Cache		MPPA® 400 MHz RDMA	
Nb Cores	FPS	Speedup	FPS	Speedup
1	3.6	1.0	3.7	1.0
2	6.9	1.9	7.4	2.0
4	13.3	3.7	14.5	3.9
8	24.4	6.8	27.4	7.4
16	40.5	11.2	49.4	13.4

Table 7.2 – *fps and Speedups for one MPPA® Cluster*

MPPA® Results The application mapping is performed at the CC level. CCs are considered as multi-core CPU to map clustered actors, and we exploit sub-graph parallelism inside CCs when possible to obtain thread-level parallelism.



(a) *Flat Mapping* **(b)** *Hierarchical Mapping*

Figure 7.6 – *MPPA® Matrix Result in fps*

Figure 7.6b plots the application performance in fps, measured when using a variable number of PEs per CC, and a variable number of CCs on the MPPA®. Using one PE in each of the 16 Compute Clusters (CCs) provides lower fps than using 16 PE of a single CC, because of the intensive usage of the local on-chip memory and Network on Chip (NoC) communications are reduced compared to the multi-cluster approach. However, in our case, this runtime overhead remains low as the parallelism is known statically. It can be noticed that performances in 7.6b for one CC are lower than the ones shown in the mono-cluster configuration of Table 7.2.

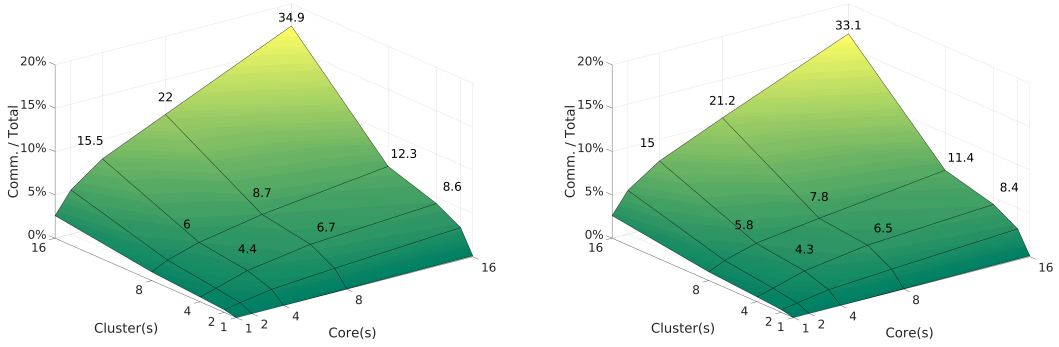
Furthermore, the IBSDF application description provides as parameters the granularity of the different level of parallelism. The idea is to have a memory allocation and scheduling (mapping) aware of the location of the memory compared to the location of the PEs. Such optimization is crucial as unnecessary data movements and the sharing of data (cache stalls on coherent architectures) will make the performances drop drastically. Parallelism is done using OpenMP 3.0 thanks to the automatically generated *omp parallel for* compiler directives on finite *for* loops. Then as the application mapping is solved statically, the number of loop iterations is known by the compiler, and thus it is easier to predict the execution time.

A total speedup of **58.7** is reached when using all 16 PEs of all 16 Compute Clusters (CCs). Although we have some scalability, we hit the memory bandwidth wall (Section 7.4.2) of manycore processors when the 256 Processing Elements (PEs) are competing for the main memory (external). Thus we focus on local memory usage at code generation to save main memory bandwidth.

7.4.2 Comparisons with Flat IBSDF Mapping

Performances Analysis For shared memory architectures Intel and TI C6678 EVM, the flat IBSDF gives the same performances as in Table 7.1. Figure 7.6a performances are lower than 7.6b by **4%** when using all processing elements of the manycore. This difference is mainly due to RDMA memory accesses coalescing, which are provided by the hierarchical mapping approach. This phenomenon is well shown by Figure 7.7a when using 8 CCs with 16 cores. The flat state-of-the-art IBSDF mapping makes each core perform RDMA transactions, which increases the ratio communication vs compute by **7.8%** concerning our new hierarchical approach.

Memory Wall in Manycore The NoC communication overhead measures the time taken by communications of all compute CCs with the main memory using RDMA. Thenceforth, the measurement considers parallel NoC communications and accumulates only the ones whose delays have an impact on the global processing time of a frame. We measure the critical communication path as the system is massively parallel, and some communications are overlapped with the computation of other CCs. In our hierarchical software synthesis method, Figure 7.7b shows where the bottleneck is when the number of PEs and CCs increases. In Figure 7.7b, the lower is the better as it shows the stalls of PEs on off-chip memory accesses (high memory access latency). In this application, NoC communications are less than 8% when using 1/4 of the processor capabilities (for instance 8 CCs with 8 PEs of the MPPA® manycore processor). Main memory accesses are starting to become significant when using more than half of the chip capability. Indeed, we have many processing elements that are competing for main memory accesses. The ratio between computation and NoC communications is higher than 30% when all PEs of an MPPA® processor are used.



(a) *Flat Mapping* **(b)** *Hierarchical Mapping*

Figure 7.7 – *MPPA® Matrix Results Ratios between NoC Communications and Processing Time (lower is better, lower means more PEs efficiency). Communication Overheads Relative to Total Execution Time.*

However, the software synthesis code exploits the on-chip local memory when chaining kernels (*sobel*, *erode* and *dilation*). It reduces significantly the pressure on the main memory that is a huge performance bottleneck (main memory bandwidth). But still, Figures 7.7a and 7.7b are crucial to analyze what needs to be optimized on manycore processors. The key to the performance (and parallel scalability) is to exploit the local memories by chaining kernels locally; otherwise, the application is always going to be IO bound [WWP09]. Nevertheless, in our example, the communication and computation ratio becomes essential when using all PEs. In the next chapters, we focus on automatic buffer prefetching, also using RDMA. Prefetching is crucial to reach the peak performance of manycores as it reduces the stall duration of PE blocked by Read-After-Write (RAW) data dependencies.

Mapping The mapping problem is NP-complete [BB07]. Its complexity increases exponentially with the numbers of PEs and actors. On a manycore with 256 Processing Elements (PEs), it becomes very complicated for both theoretical mapping algorithms and their implementations. In our case, once the application parallelism is revealed by applying

the flattening and single-rate transformation to all hierarchical actors, the resulting graph contains more than 1,000 actors and 800 edges to be mapped on 256 Processing Elements (PEs). Thus, the flat IBSDF graph is scheduled and mapped on the processing elements in **26 minutes**. With the hierarchical mapping approach, the process lasts less than **one second**. On more complex applications, for instance, use cases with more than 10,000 actors, the hierarchical approach is a must-have feature as the mapping time explodes.

7.5 Conclusion

In this chapter, we introduce a new technique to exploit both coarse-grained and fine-grained parallelism based on a hierarchical dataflow programming model. The main advantage of this strategy is that it provides the transformation workflow with scheduling and code generation simplifications. Our strategy also improves data locality, which is crucial for high-performance and power consumption. Indeed data movements have a significant impact concerning time and energy, especially for embedded MPSoCs.

The fine-grained parallelism is retrieved by applying *omp parallel for* onto Repetition Vectors (RVs) automatically extracted in a hierarchical actor. We show that this approach matches not only manycore processors with a distributed memory architecture but also multi-core architectures with shared memory.

In the future, System-on-Chip (SoC) will embed more and more heterogeneous PEs and memories. Therefore, the mapping on such architectures will become more and more complex. In our example, we used a low-level image processing application and show significant speedups when the number of PEs increases. The mapping of an application is not a simple problem, and it is becoming more and more involved with increases in architectural complexity (number of PEs, PE heterogeneity within the same SoC, memory hierarchy, hardware accelerators). The hierarchy of dataflow programming models is one of the key assets to program complex architectures like the Kalray MPPA® manycore processor.

CHAPTER 8

Porting an Embedded Runtime for Executing Reconfigurable Dataflow onto a Clustered Manycore Processor

As shown in Chapter 7, dataflow models can be used at compile time to ease the programming of manycore processors. The programming model used in this work at compile time is the Interface-Based SDF (IBSDF), and its hierarchical semantics. IBSDF interfaces are inherited by the Parameterized and Interfaced dataflow Meta-Model (PiMM) meta-model, and its application to the Synchronous Dataflow (SDF) programming model, called Parameterized and Interfaced SDF (PiSDF) [DPN+13]. PiSDF extends the semantics of IBSDF by introducing explicit parameters, and a parameter dependency tree. The primary goal is to increase the expressivity of the IBSDF, and thus, to model advanced real-life applications in which much control and decisions have to be handled at runtime. Compared with dynamic dataflow semantics, the PiSDF maintains strong predictability, enforces the conciseness, and readability of application descriptions.

The parameters introduced in the PiSDF programming model can be modified at runtime. The dedicated runtime called Synchronous Parameterized Interfaced Dataflow Embedded Runtime (SPIDER) has been developed to execute an application efficiently, described using the PiSDF model, as seen Section 3.5.2. Paper [HPD+14] shows that SPIDER outperforms Open Multi-Processing (OpenMP), and the dynamic dataflow compiler Open RVC-CAL Compiler (Orcc), proving that the PiSDF offers an excellent trade-off between dynamicity and predictability. The original implementation of the SPIDER runtime supports shared memory based Multiprocessor Systems-on-Chips (MPSoCs), and experiments have been done on Intel® Central Processing Units (CPUs) and multi-core Digital Signal Processors (DSPs). Supporting shared memory architectures on the recent PiSDF programming model was challenging and a necessary milestone in this research.

We show in this chapter how we have adapted the SPIDER runtime for executing applications described in PiSDF onto clustered manycore machines. The development part of this work has been done by Hugo Miomandre during his final year internship under the supervision of Karol Desnos and myself for debugging. The internship was partially supported by the MORDRED project, funded by the GdR ISIS of the CNRS.

Shared-memory architectures are easier to use than distributed memory architectures models thanks to their global address space, and the provided hardware synchronization mechanisms (usually atomics). The key challenges to target the Multi-Purpose Processor Array (MPPA)® and more generally clustered manycore architectures are listed below.

- Initializations of more than 256 Processing Elements (PEs)
- Sharing of resources
- Finalizations and exits
- Memory allocation (distributed and shared memories)
- Synchronizations (distributed and shared memories)
- Communications (distributed and shared memories)

The proposed extension supports Direct Memory Access (DMA)-enabled architectures implementing *One-Sided* communications developed during this thesis and described in Chapter 5. The original implementation of SPIDER is designed and partitioned as follow: graph modeling, graph transformation, graph scheduling/mapping, memory allocation, synchronizations, and communications. We will see in this chapter how both the architecture independent front-end and the architecture dependent back-end have been modified to make it fit the Kalray MPPA® manycore processor.

This chapter is organized as follows. Section 8.1 presents the architecture of the SPIDER runtime. Optimization heuristics regarding the scheduling are explained in Section 8.2. The management of the distributed memory and explicit memory communications are presented in Section 8.3. Finally, Section 8.4 contains results, explains and compares them with previous results given in Chapter 7.

8.1 Architecture of the Distributed Dataflow Runtime

SPIDER operates as an offloading runtime similar to OpenCL or OpenMP 4.0. The main application offloads computations on the acceleration cores. The dataflow runtime has a master/slave approach.

As seen in Section 3.5.2, the master process needs a PiSDF graph description of the application generated by Parallel and Real-time Embedded Executives Scheduling Method (PREESM) and distributes at runtime the computation on the slave PEs, called Local RunTime (LRT).

Such a model offers several advantages such as a centralized control, the ability to trigger the offloading of any dataflow graphs, depending on external events, and to manage error recovery on complex parallel systems.

We contributed two ports. The first one uses shared memory, which runs on a single Input/Output Subsystem (IO) of the MPPA® processor. The second one uses 16 Compute Clusters (CCs) and one IO of the MPPA® processor, and it performs automatically explicit DMA communications to handle the distributed local memories of the clustered manycore architecture.

The first porting step consisted in compiling the original SPIDER runtime for the Kalray Very Long Instruction Word (VLIW) core using only the bare-hypervised toolchain. Then, using the *pthread* Application Programming Interface (API), the SPIDER runs on the main program thread, and up to 3 Local RunTimes (LRTs) running on 3 threads are possible using the multi-threading runtime presented in Chapter 6 onto the Input/Output Subsystem (IO) of MPPA®.

Such support was not as easy as it is on a Linux system because of the stack size limitation, unsupported low-level functions, that needed to be bypassed or reimplemented, differently and the heterogeneous memory map of the Input/Output Subsystem (IO) (two

local memories and one Double Data Rate (DDR)). However, thanks to the contribution of Chapter 6 it was possible.

8.1.1 Insight of the Global MPPA® Implementation

The implementation of the distributed SPIDER runtime is more complicated than the shared memory implementation concerning the deployment and the execution part. But still, the shared memory implementation was a necessary step in the development. Figure 8.1 shows the overall architecture of the reconfigurable dataflow runtime onto a full Kalray MPPA® processor.

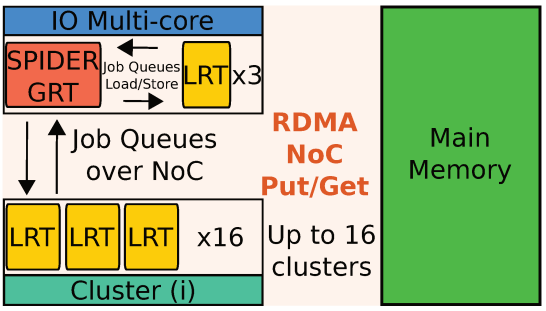


Figure 8.1 – *Architecture of the Reconfigurable Dataflow Runtime onto a DMA-Enabled Clustered Manycore Processor*

The SPIDER (Global RunTime (GRT)) runtime is mapped onto one core of the Input/Output Subsystem (IO) that is in charge of booting and controlling the Compute Clusters (CCs), initializing the platform data structure, managing the main memory (Figure 8.1), handling graph transformation (reconfigurations), mapping/scheduling the *Single-Rated* graph, and sending ready tasks through the job queues to the LRTs (cores).

As in the shared memory implementation, the IOs also implements up to 3 LRTs that can process tasks. Those 3 LRTs are useful when the application embeds actors that both consumes and produces more than the available local memory within the Compute Cluster (CC). However, tasks with large memory footprints are mapped on the Input/Output Subsystem (IO), which has direct memory access to the 4 gigabytes of memory of the DDR memory.

All mechanisms presented in the porting of the runtime for the DMA-enabled processor are fully automated in the SPIDER, and transparent to the application developers. Hence, the application developer specifies the application PiSDF graph in the PREESM tool, the computation kernels associated with the PiSDF actors, and accesses to input and output buffers of the application through data pointers. In particular, the data movements represent a significant part of the work. They are managed implicitly by the distributed dataflow runtime. Indeed, the new SPIDER runtime automatically performs all the required communications explicitly, using the underlying low-level Asynchronous One-Sided (AOS) communication API, part of the Kalray toolchain, and introduced in Chapter 5.

8.1.2 Structure of the Original Synchronization Protocol

Each actor of the PiSDF graph implements a task to be run sequentially on a PE of the targeted platform. The original SPIDER synchronization protocol used by LRTs to trigger the execution of actors implements the following sequences.

1) When an LRT completes a task, the PE running the LRT writes the produced data tokens into shared memory. On completion of the task, the LRT sets its job ID to the completed job ID. A job ID is a 64-bit monotonic counter that identifies the task that is fired on an LRT. The job ID is used to synchronize PEs with each other during the execution. Therefore, the job ID is written in the corresponding slot (related to the PE), in an array of slot in shared memory, precisely the *array[LRT number]* slot. Each slot has multiple readers and a single writer.

2) Once this LRT is available for further computing, it tries to get a new job from its task job queue.

3) Before the firing of the popped task, the LRT must wait for the completion of preceding tasks. Such constraints are the native data dependency of the application. For that purpose, job messages contain the job ID of each preceding task and the ID of each LRT that executed these jobs. Thus the LRT will compare the expected job counter values, given by the job IDs, and the actual job counter values of the specified remote LRTs.

For convenience, job counter values of all LRTs are stored in a single array explained in (1), accessible as Read-Only to all PEs. Such synchronization mechanisms are quite simple to implement with shared memory architectures. Indeed, the access to these job counters and IDs are performed by *Load/Store* in the shared memory, respecting the memory consistency rules of the targeted architecture. As already seen in Chapter 4.4, the memory consistency is guaranteed by the special full memory barrier instruction to prevent data races.

A Word on the SPIDER Implementation for the Texas Instruments (TI) Keystone II On previous SPIDER implementations [HPD+14], designed for the PEs of the TI Keystone II architecture, this synchronization mechanism uses hardware queues to manage data dependencies. The TI Keystone II implements many hardware queues, making the synchronization of large dataflow graphs possible (few thousands of vertices at a time). Such hardware specific implementation shows that the SPIDER architecture provides a proper partitioning of the key actions of the runtime, and allow these actions to be accelerated by hardware specific features of the targeted platform.

8.1.3 Implementation of a Distributed Synchronization Protocol

The objective of the new synchronization algorithm is to both distribute the control of synchronizations and bound the number of Network on Chip (NoC) communications per data dependency necessary to fire an actor (run its associated task sequentially).

The proposed algorithm built on the "observer design pattern" [Gam95], where the *observers* are the LRTs waiting for the completion of a preceding actor and the *notifier* is the LRT executing this actor. The operating principle of the algorithm is based on three key actions:

Register: When an LRT pops a new job from its queue, it scans the set of preceding actors in the job descriptor (sent by the SPIDER GRT running on the Input/Output Subsystem (IO)), and sends a *notification request* to each LRT executing the preceding actors. A *notification request* encapsulates both the ID of its sender LRT, and the awaited job ID.

Notify: On job completion, an LRT updates its *job counter*, then process all its pending *notification requests* with an awaited ID lower than the new *job counter* value. Software flow control is performed to avoid data corruption on congestion.

Peek: Optionally and for optimization purpose, after sending all its *notification requests*, an LRT can check, once and on its own, the *job counter* values of all LRTs that have not yet answered.

The goal of a peek, which consists of a remote 8-byte load in a remote memory over the NoC, is to avoid waiting for a notification from a busy LRT whose *job counter* is already greater than the awaited value.

The remote 8-byte load is part of the set of primitives provided by the Asynchronous One-Sided (AOS) distributed communication API. As the latency of the *Peek* operation is high, typically more than a thousand of cycles (see Figure 5.11), the written transaction performs several asynchronous calls for the overlapping of as many transactions as possible (along with the coalescing of information within bit-fields).

Using these actions, each data dependency requires at most five communications through the NoC: two to send a *notification request*, one to send a notification, and two for a *Peek*. Hence, a finite number of NoC communications per dependency is needed, which fulfills the communication bounding goal.

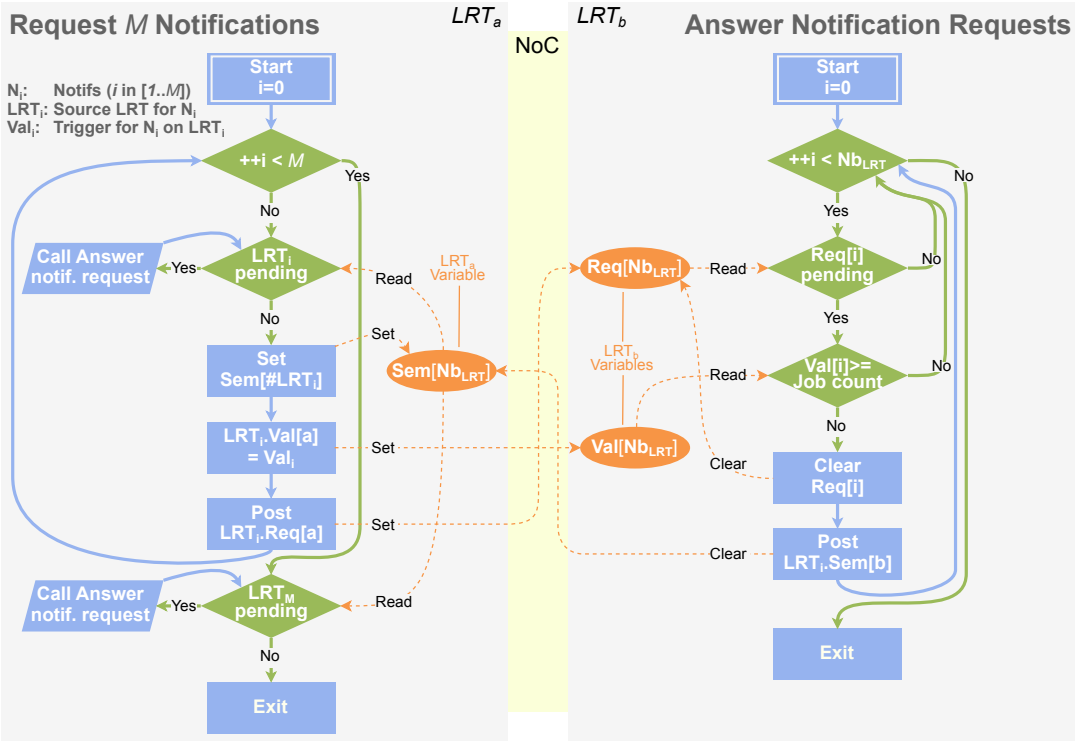


Figure 8.2 – *Algorithms for Distributed Synchronizations for the Actor Firings. The number of requests is the number of input First-In-First-Out queues (FIFOs) of the next actor.*

The algorithm flow-chart in Figure 8.2 details the distributed synchronization protocol. The protocol implements an all-to-all (LRTs) synchronization mechanism. The left part of the diagram describes the observer LRT_a popping a new job from its *job queue* and the right part explains the notification sequence when LRT_b processes pending notification requests. To simplify the figure, the *peek* action was omitted. The protocol requires three synchronization vectors for each LRT, allocated in the local memory of the PE: *Sem* contains the LRTs IDs of sent but pending *notification request*. *Req* registers the LRTs IDs of received *notification requests*. *Val* contains the job counter values awaited by LRTs

registered in *Req*. The size of each array corresponds to the total number of LRTs in the system Nb_{LRT}.

8.2 Optimized Heuristic-based Scheduling

This section focuses on mapping and scheduling. It is complicated to be handled at runtime when using the commonly used methods. The Section 8.2.1 defines the problem and its bottleneck, and Section 8.2.2 describes a solution using a simple and efficient mechanism when targeting massively parallel architectures.

8.2.1 Prohibitive Complexity and Footprint

The original scheduler implemented in SPIDER is a LIST scheduling heuristic described in [Kwo97]. When the input parameters of a dataflow graph are set dynamically, the GRT analyzes the data exchange rates in the PiSDF graph and generates an equivalent *Single-Rate* Directed Acyclic Graph (DAG) graph, exposing explicitly all data parallelism. Actors of the DAG are obtained by duplicating actors of the PiSDF graph as many times as their number of firings; themselves obtained analytically from data consumption and production rates [LM87]. Then, the GRT handles the mapping and scheduling of each actor, taking into account the dependencies of the DAG and mapping constraints if any. A mapping constraint can be some user defined task assignments to specific PEs, and the local memory usage on clustered architectures.

The issue, with the LIST scheduler, is that its complexity becomes prohibitively large when targeting a processor with hundreds of PEs. Indeed, it is $O(A.log(A) + P.(A + E))$ [Kwo97], where A and E are the number of actors and dependencies in the DAG (vertices and edges), and P is the number of cores. As already seen, manycore architectures implement hundreds of cores and require many parallelisms to be useful. Therefore, the number of DAG actors to be scheduled in parallel increases roughly linearly with the number of PEs. Consequently, the complexity of the LIST scheduling increases quadratically with the number of PEs, making it a bottleneck for runtime scheduling.

8.2.2 Lightweight Scheduling, Simpler is Faster

We replaced the original LIST scheduler with a less complex scheduling algorithm based on a specialized Round Robin (RR) heuristics. Firstly, the new algorithm was designed to reduce the memory footprint and the latency of job scheduling decisions. The main idea is to increase as much as possible the dispatch rate of the job, in other words: the performance in Input/Output Operation per Seconds (IOPSs) of the GRT (scheduler). The classical RR heuristic iterates circularly on a list of LRTs, and sends jobs to the first available LRT.

This heuristic lowers the scheduling complexity down to $O(A + E)$, as a topological ordering of actors is required. However, we found that the evaluation of the actor execution time and the job fairness distribution to LRTs were no longer required, as a lot of LRTs are available, and, at least one LRT is always ready to compute.

Secondly, memory usage is optimized by interleaving the PEs from different CCs in the list on which our RR algorithm iterates. In each Compute Cluster (CC), 16 PEs share a local memory and a NoC interface. The goal is to prevent too many jobs from being sent simultaneously to PEs on the same CC. As such, it provides higher on-chip memory usage and reduces the pressure on the local memory of the CC. Indeed, multiple tasks starting their execution try to synchronize themselves with their predecessor actors, and allocate

local memory in the CC for their input and output buffers (see Section 8.3). Therefore, the scatter of jobs among CCs prevents and reduces the wait time to access shared Compute Cluster (CC) resources.

Thirdly, the specialized RR algorithm uses a flow control mechanism to avoid overflowing job queues. The GRT sends a job to an LRT only if the remote job queue contains enough space to receive the job. For that purpose, LRTs send their job counter value to the GRT on job completion. The statically configurable size of *job queue*s has to be large enough to prevent starvation of the LRTs, but small enough to keep the memory footprint under control.

Finally, the RR scheduler was tuned to take into account the available memory in the Compute Cluster (CC). When the task of an actor is scheduled, the required amount of memory for its execution is computed, and the scheduler does not send it to an LRT that is not able to run it due to the lack of local memory. The measurement of memory space available in the local memory of the CC, associated with each PE, is done off-line but can be performed at runtime, while initializing the SPIDER.

8.3 Managing the Distributed Memory at Runtime

Section 8.3.1 explains issues encountered when dealing with the array of local memories of the MPPA® processor. The MPPA® architecture implements neither global cache hierarchy nor cache coherence at any levels. As such it is challenging to control and move data explicitly by software, which is also pointed out in the paper [WWP09] regarding DMA-enabled processors. Section 8.3.2 explains the memory allocator in each Compute Cluster (CC), and the handling of the corner and unsupported cases (deadlock, congestion).

8.3.1 Distributed Local Memories instead of Caches

Once an LRT pops a new job, it needs to allocate memory to accommodate the input and output buffers of the corresponding actor. As the original SPIDER was implemented for shared memory architectures, where PEs (LRTs) access the main memory, usually DDR technology in embedded systems, through their data cache, the SPIDER GRT uses a single global memory allocator. Data pointers on globally allocated data tokens of the FIFOs are sent to the LRT job queues, and LRTs can access data tokens using *Load/Store* instructions. The hardware data cache does the communication automatically (implicitly) by refilling the requested data from the main memory into the cache close to the PEs. However, memory consistency operations at synchronization points are still required (full memory barrier).

On a local memory based manycore architecture, the memory in the multi-core Compute Clusters (CCs) needs to be allocated by each local runtime, as well as the movement of data. Once again, the movement of data is performed by the Asynchronous One-Sided (AOS) communication API, using the Remote Direct Memory Access (RDMA) *Put/Get* protocol as shown in Figure 8.1.

Linear (contiguous) *Get* operations are used to read the input FIFOs of the actor from the DDR to the on-chip local memory of the CC. When the execution of the scheduled actor completes locally in the Compute Cluster (CC), linear (contiguous) *Put* operations are initiated by the LRT to write back the data in the DDR memory. RDMA fences are then issued to get the completion of the multi-CC Read-After-Write (RAW) data dependency.

Furthermore, distributed local memories are limited, thus making the control software more complicated and error-prone. In such a case, memory resources may be exhausted

frequently, but do not necessarily impose the termination of the application. For instance, a memory allocation may fail for an actor A when another actor B, executed in the same CC, uses all the available local memory. On completion of the execution of actor B, its memory can be reused, possibly after sending output buffers back to the main external memory (off-chip). Then actor A may successfully perform its memory allocation in the local memory.

8.3.2 Thread-safe Local Memory Allocator

The flow-chart in Figure 8.3 describes a new algorithm to allocate space in the local memory of a clustered manycore architecture. Such allocation procedure ensures that all scheduled jobs on an LRT running in a CC manage to allocate their required memory, as long as it does not exceed the maximum capacity of the local memory space of the CC (see Section 8.2). When the firing conditions of a mapped actor are fulfilled, the LRT attempts to allocate its buffers using the algorithm in Figure 8.3.

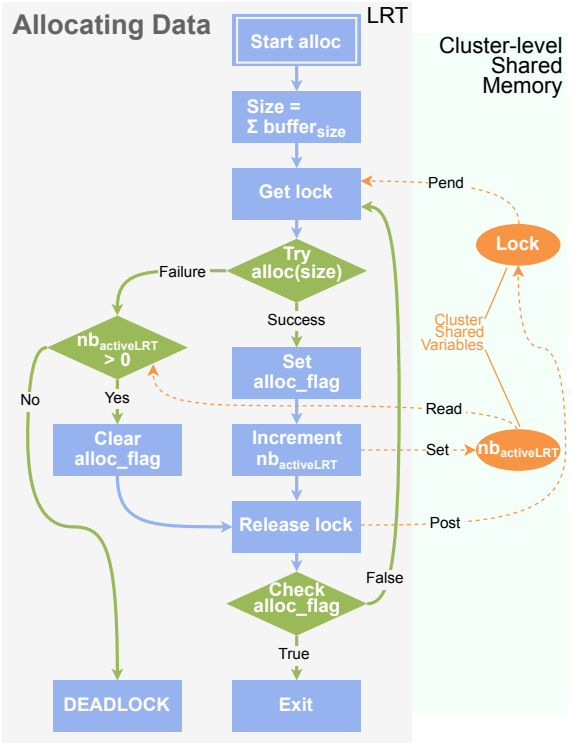


Figure 8.3 – *Algorithm for the Local Memory Allocation in the CC.*

As multiple cores may compete for local memory space, a CC level *lock*, based on atomic instructions, is required to prevent the memory allocator from data structure corruption. The critical section of this algorithm also protects a shared counter, $Nb_{ActiveLRT}$ that represents the number of actors (task) currently executed on the Processing Elements (PEs) concurrently within the CC. If the number of active LRT is greater than zero and a memory allocation fails, the LRT should release the *lock*, and try again later. If not, a deadlock is detected as no other LRT is currently using CC memory, and there is no reason for more memory to be available during a future allocation attempt. The deadlock detection is an expendable safety feature if, as presented in Section 8.2, the scheduling

algorithm is aware of the maximum available space in CC memories. It is also used to debug the runtime on failure.

The deallocation procedure on actor completion takes the *lock*, frees the buffers, decrements the $Nb_{ActiveLRT}$ counter, releases the *lock*, and performs a full memory barrier to guarantee the cache coherence in the CC.

8.4 Results and Comparisons

The image processing PiSDF graph used to validate the SPIDER manycore implementation is presented in Figure 8.4. The use-case is a flat refinement of the previously described application in Chapter 7 (see Figure 7.1). However, the use-case in this chapter implements a dynamically reconfigurable parameter N within the dataflow. In Figure 8.4, notations $*N$ next to the actors denotes the number of (parallel) executions of each actor during a graph iteration. In this example, the *Sobel*, *Dilation* and *Erosion* actors are parallel, and the other actors are sequential.

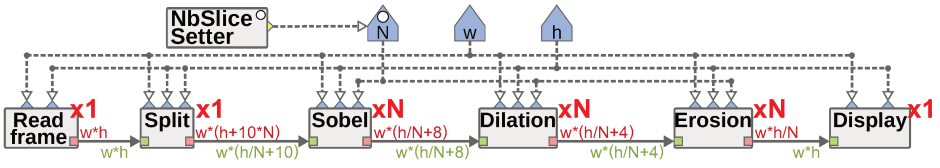


Figure 8.4 – *Parametrized Image Filtering Application.*

The reconfigurable parameter N represents the number of slices the input image. In this example, actor *NbSliceSetter* computes the value of N, that maximizes application performance according to the image size. The selected application demonstrates the feasibility of a reconfigurable dataflow runtime on a manycore architecture. The runtime overhead is evaluated in comparison with a static execution.

8.4.1 Memory Footprint of LRT

A first porting of the SPIDER LRT runtime on MPPA® PEs resulted in a code and data footprint of 82 kilobytes. Mapping an LRT process on each of the 16 PEs of a CC requires about 1.28 megabytes out of the 2 megabytes of local memory. Considering that about 620 kilobytes of memory is reserved for system services, only 116 kilobytes of memory remains available to store application data. 116 kilobytes makes the execution of the filtering application possible on image resolutions up to *720p* with 256 slices processed in parallel.

Thanks to the scheduling mechanism limiting the number of jobs sent to individual LRTs, the size of the *job queues* is reduced to three slots, enabling job buffering and leading to an LRT memory footprint of 6.5 kilobytes. In this configuration, 1.29 megabytes of CC memory remains available for storing processed data. This new configuration leaves enough space to allow the processing of ultra high-resolution (4K) images, that is precisely a resolution of $3840 * 2160$, on the MPPA® processor of the second generation.

8.4.2 Performance, and SPIDER Overhead

Figure 8.5 shows the Frames per second (fps) obtained when executing the image filtering PiSDF graph on the MPPA® for 4K images. They are plotted for different numbers of active CC, and different numbers of active PEs per CC. The sequential performance on a

single PE is 0.13 fps. When 256 PEs are used, a throughput of 2.81 fps is reached, which represents a speedup of 22 compared to the sequential execution. During the processing of each frame (0.36s), only 8% of the time is due exclusively to GRT computations. Hence, actor computations and NoC communications are responsible for 92% of it.

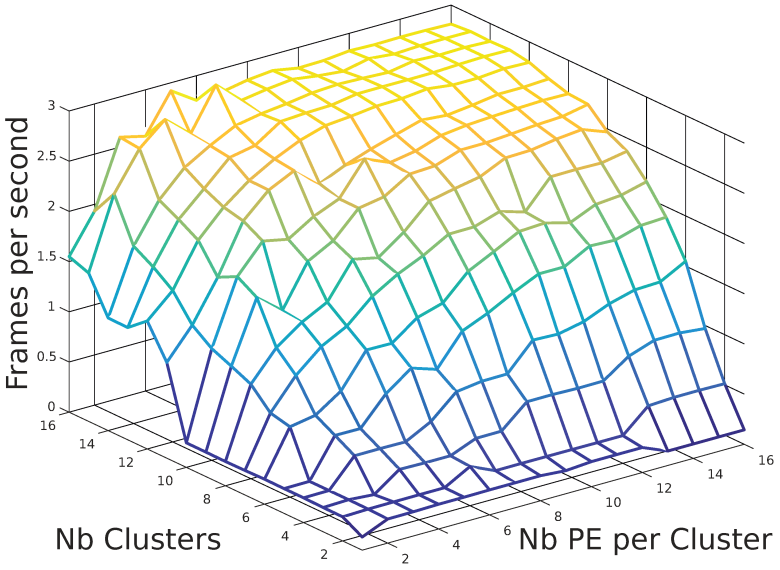


Figure 8.5 – *Application Performance on a 4K Video*

These results are expected since, according to Amdahl's law, the theoretical speedup for this application on 256 Processing Elements (PEs) is 28, which is an optimistic prediction as the communications overhead is ignored.

The Amdahl's law is well known and commonly used to define the efficiency of a parallel implementation. The Amdahl's law defines the speedup limits of an application that can be parallelized. Indeed, when parallelizing an application, there is often a sequential part of it that cannot be parallelized. Because of that, the theoretical speedup is bounded and given by $speedup = \frac{1}{(1-p)+\frac{p}{s}}$. p is the sequential part of the program that can be parallelized. s is the actual speedup of this part p.

Calculation of Amdahl's law is based on a measurement of T_{par}, the summed execution time of the image processing actors (Sobel, Dilation, Erosion) on the Compute Cluster (CC), and T_{seq} the summed execution time of all other actors on the Input/Output Subsystem (IO). The theoretical speedup S on 256 PEs is given by $S = ((1 - \frac{T_{par}}{(T_{seq}+T_{par})}) + \frac{T_{par}}{256\times(T_{seq}+T_{par})})^{-1} = 28$.

When using a standard thread-based implementation of SPIDER on an Intel Xeon E5-1650 with 6 hyper-threaded x86 cores clocked at 3.60 GHz, the processing of a 4K video, with the same PiSDF graph, reaches 11.40 fps, using 95% of all CPU time. Although the performance on the Xeon processor is almost 4 times better, this processor dissipates on average 10 times more power than the MPPA®. Hence, the execution on the MPPA® is approximately 2.5 more energy-efficient than on the Xeon.

Comparison with Previous Work on the MPPA® Processor In Chapter 7, the measured performances of a static mapping of the PiSDF graph from Figure 8.4 are better. In the static version, N is fixed, and all mapping and scheduling is performed at compile time for VGA videos ($640 * 480$).

The top performance obtained for the static execution is 217 fps. For this video resolution, the reconfigurable PiSDF graph, executed with SPIDER, peaks at 47 fps. Besides the SPIDER runtime overhead, the difference between the performance of the static and reconfigurable executions are mostly due to the lack of memory optimization in the reconfigurable implementation (dynamic). In the reconfigurable version, many copy calls (*memcpy* in C) are performed to create the image slices in the *Split* actor and to merge processed slices into a contiguous buffer before *Display*. Thanks to compile-time optimizations, these *memcpy* calls are replaced with pointer and DMA offset operations in the static version reducing the memory transfers by a factor of 3 (memory bandwidth reduction of the memory accesses).

8.5 Conclusion

This chapter presents an implementation of a runtime manager that leverages reconfigurable dataflow graphs on manycore architectures. To the best of our knowledge, this is the first online mapping, and scheduling of a parametric dataflow application onto a clustered manycore architecture.

At first, we ported the runtime on a single multi-core CPU, namely the Input/Output Subsystem (IO) of the MPPA® for legacy. We then expanded the runtime globally on the MPPA® processor. The master GRT runtime operates on the IO and the slaves on the CC. For that, our runtime supports distributed memories and manages explicit cores data communications using RDMA, queues, and remote atomic operations for synchronizations. Furthermore, some scheduling methods based on efficient heuristics are introduced to let the master runtime feed all the slaves mapped on the Processing Elements (PEs) of the Compute Clusters (CCs). New memory allocation algorithms were also specifically designed to provide efficient usage of the on-chip memory and for catching memory allocation errors, if any, as memory is a critical resource on manycore processors with local memories. Experiments on the Kalray MPPA® processor demonstrate the feasibility of such a runtime, its potential concerning application performances, and energy efficiency.

However, such distributed runtime was challenging to implement, debug and validate. Indeed, the highly concurrent environment, with several multi-core CPUs, and with different memory maps is difficult to analyze. For instance, the implementation of flow control, the lack of hardware memory coherency, the management of explicit communications and memory allocation were all challenging. However, the final solution is today operational with high software maturity.

CHAPTER 9

A Distributed OpenVX Framework for a Clustered Manycore Processor

As already seen in previous chapters, the programming of Direct Memory Access (DMA)-enabled processors is challenging and difficult [WWP09]. To make it easier, this chapter describes the first OpenVX implementation for the Kalray Multi-Purpose Processor Array (MPPA)® processor.

OpenVX [G+17] is a standard developed by the Khronos group for cross-platform acceleration of computer vision and deep learning applications. It is a domain specific Application Programming Interface (API) (like a Domain Specific Language (DSL)) that abstracts the architecture complexity (heterogeneity) of the processor. Moreover, OpenVX is a serious candidate for application engineers who need high-performance embedded software for vision and learning applications. All optimizations are performed automatically by the proposed framework for a wide range of application kernels. OpenVX aims to be at a much higher level than other standards such as OpenCL, OpenMP or academic models like Interface-Based SDF (IBSDF) or Parameterized and Interfaced dataflow Meta-Model (PiMM), which require a clear understanding of the application and its manual parallelization. OpenVX vendors perform all optimization work to make the application run efficiently.

The Khronos OpenVX standard [G+17] uses a graph-based approach to ease the design of computer vision pipelines and decrease the time to market. The graph-based computing may optimize sequences of kernels at graph level to get the best out of the hardware capabilities. The graph-level approach makes it possible to optimize at high-level of sequences of kernels, and at low-level, to use stream memory accesses or use vector instructions (Single Instruction, Multiple Data (SIMD)). Both low-level and high-level optimizations are up to the OpenVX vendors.

In the computer vision domain, open-source libraries, like OpenCV, are also designed for rapid prototyping onto general purpose parallel processors. However, in these libraries, the computation is performed explicitly at each function call; therefore, it is impossible to perform global optimization over several functions. For instance, it is impossible to group kernels to increase data locality. Such optimizations are of utmost importance for performance optimization as it reduces the memory traffic, a bottleneck in most High-Performance Computing (HPC) systems.

Our OpenVX implementation targets low latency and parallel graph execution to enable reactive embedded systems. Each compute-intensive kernel is distributed on the en-

tire Compute Cluster (CC) matrix of the MPPA® processor. Low latency implementations (also called batch-1) are very different from high throughput implementations. High throughput implementations are usually based on the pipelining of the graph execution, which is not very complicated. In our case, we do not use graph pipelining optimization. Instead, we distribute each node on all available computing resources. We automatically perform the scheduling, the memory allocations, and the data transfers to satisfy multi-cluster Read-After-Write (RAW) dependencies.

Our OpenVX framework has been written from scratch, starting from the specification and the Khronos API provided in *https://github.com/KhronosGroup/ OpenVX-Registry*. The implementation is based on the multi-threading runtime and the asynchronous one-sided API that are both presented in Chapters 5 and 6 respectively.

Section 9.1 presents the OpenVX standard, third-party implementations, and compares OpenVX with OpenCL. Section 9.2 explains the back-end for offloading computations from the Input/Output Subsystem (IO) to the Compute Clusters (CCs) of the MPPA® processor. The dynamic optimization is described in the *vxVerifyGraph*, Section 9.3

9.1 Requirements and Positioning

In Section 9.1.1, we explain the main ideas of the OpenVX standard such as the different objects, and the architecture itself. In Section 9.1.2, we describe existing third-party implementations that are either academic or commercial. Implementations details are given as well as the design strategies and runtime dependencies. Section 9.1.3 explains differences between OpenCL and OpenVX.

9.1.1 OpenVX Standard and Example

The OpenVX standard [G+17] is a graph-based API designed by the Khronos group for developing computer vision and deep learning applications on embedded platforms. The standard is usually implemented and proposed by hardware manufacturers in their programming environments. OpenVX is not only designed for a Central Processing Unit (CPU)-Graphics Processing Unit (GPU) target like OpenCL but is also reminiscent of dataflow programming models. Indeed OpenVX has already shown its efficiency for the programming of a host associated or not with remote computing resources like Nvidia® GPUs or FPGA using CUDA® or OpenCL respectively.

As seen in Section 3.3, dataflow programming models are architecture-agnostic, highly valuable for exposing high-level optimization opportunities and enabling automatic deployment of applications on a wide variety of embedded platforms [LM87]. The OpenVX programming model is a Single-Rate (SR) specialization of the Synchronous Dataflow (SDF) programming model [BML99, LM87] where production and consumption rates of the graph nodes (the actors) are equals. So a specific strength of OpenVX is to expose the graph structure of the entire processing pipeline, to enable implementations to perform high-level optimizations, and to allow vendors to get the most out of their machines.

Figure 9.1 shows an example of OpenVX code. In this example, we removed error checks to simplify the code, but a real application would check the return values by the *creations*, *verify*, *releases* and *process* OpenVX functions.

A *context* describes the accelerator device where the computation is going to be offloaded. The standard includes very few platform-specific functions and data structures. One of these functions creates the platform description using as an input a platform-specific structure. The structure *platform* at Line 2 of the code Figure 9.1, described the config-

uration for the Kalray MPPA® processor. This platform-specific object is explained in Section 9.2.

```
1.  vx_uint32 width = 1920, height = 1080; // Full HD image
2.  vx_context context = vxCreateContextFromPlatform(&platform);
3.  vx_graph graph = vxCreateGraph(context); // Creation of the graph
4.  vx_image images[] = { // Creation of images
5.      vxCreateImage(context, width, height, VX_DF_IMAGE_U8), // Real
6.      vxCreateVirtualImage(graph, 0, 0,  VX_DF_IMAGE_U8), // Virtual
7.      vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8),  // Virtual
8.      vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8),  // Virtual
9.      vxCreateImage(context, width, height, VX_DF_IMAGE_U8), // Real
10. };
11. vx_node nodes[] = { // Create the graph or pipeline of kernels
12.     vxGaussian3x3Node(graph, images[0], images[1]),
13.     vxSobel3x3Node(graph, images[1], images[2], images[3]),
14.     vxMagnitudeNode(graph, images[2], images[3], images[4]),
15. };
16. vxVerifyGraph(graph); // Graph verification and compilation
17. vxuFReadImage(images[0], in_fd);  // Read input image
18. vxProcessGraph(graph); // Graph execution
19. vxuFWriteImage(images[4], out_fd); // Write output image
20. for (int i = 0; i < dimof(nodes); i++)
21.     vxReleaseNode(&nodes[i]); // Delete nodes
22. for (int i = 0; i < dimof(images); i++)
23.     vxReleaseImage(&images[i]); // Delete images
24. vxReleaseGraph(&graph); // Delete graph
25. vxReleaseContext(&context); // Delete context
```

Figure 9.1 – *Example of an OpenVX Application.*

Once the *context* is created, the OpenVX *graph* is created with a direct reference to the parent *context*. This is required for the static optimization passes during the *vxVerifyGraph* function. The compiler needs to know the platform to perform the proper optimization choices. The OpenVX *graph* is composed of *vertices* and *edges*. The *vertices* are called OpenVX *Nodes*. The developer can select nodes in a list of standard kernels [G+17], and supported by the platform thanks to the vendor. The *edges* correspond to OpenVX buffers (*Images, LUTs, Arrays, and Pyramids* for instance) and link the *vertices* which produce and consume data.

Two kinds of buffers exist: the user buffers, allocated and accessible from the memory space of the OpenVX host application; and the virtual buffers, that contain data exchanged between the *vertices* of the graph. Virtual buffers cannot be accessed by the host application, and they may be suppressed by using kernel fusion optimization techniques. In the code of Figure 9.1, only the input and output images are user buffers (Lines 5 and 9).

Once the graph is created, it has to be verified and compiled. To do so, the user explicitly calls the *vxVerifyGraph* function. On success, the user calls the *vxProcessGraph* function to execute one iteration of the OpenVX graph explicitly. In OpenVX, there exists functions to access the data of the OpenVX objects (*Images and Arrays* for instance). Instead, we chose to add custom vendor-specific nodes to manage the Input-Output of the graph, namely *vxuFReadImage* and *vxuFWriteImage*. They respectively fill up the input image and write the output image data to the hard-drive disk or a stubbed display function over Peripheral Component Interconnect Express (PCIE) [Aja09].

9.1.2 Third Party Implementations & Optimizations

An implementation of the OpenVX standard is provided by all Intellectual Property (IP) and chip vendors who target computer vision applications. OpenVX implementations are also available from GPUs and Field-Programmable Gate Array (FPGA) vendors, and they use the offloading foundations of CUDA®or OpenCL programming models.

Seminal OpenVX optimizations techniques are described in [RVD+14]: Inter-Process Communication (IPC) aggregation, pipelining, data prefetching, SIMD execution, and multiple levels of block tiling. All of these optimizations are the basics of efficient parallel implementations [WWP09], and they are applied to third-party OpenVX frameworks presented in this section.

The Nvidia® VisionWorks framework, presented in [BA14] implements OpenVX using CUDA for GPU offloading. The Advanced Micro Devices (AMD) open-source framework described in [GP16] uses either OpenCL for GPU offloading or the host CPU for computations. AMD kernels use SIMD and streaming *Load/Store* to access sparse data at CPU level efficiently. Also, they use OpenCL to offload kernels onto GPU when available. Both frameworks target GPU-based accelerators or the host processor.

The ADRENALINE framework presented in [THB14] [THMB15] features a series of optimization techniques including kernel fusion, overlap tiling by recomputing halo regions (ghost regions [KS10]), and double buffering for overlapping computation and communications. ADRENALINE provides a virtual prototyping platform, currently implementing a single cluster and a host CPU. Their runtime is built on OpenCL 1.1 [G+11] with an extension to exploit the on-chip memory efficiently, avoiding round trips to the main memory (external) whenever possible.

By comparison with ADRENALINE, our work focuses on OpenVX graph optimizations in a standalone mode (without external CPU), and targets low latency execution times using multiple clusters. The standalone mode let our framework compile on-the-fly, with a call to *vxVerifyGraph* at runtime, the OpenVX graph onto the target processor, when a configuration parameter of the OpenVX application changes. We instantiate a multi-core host CPU on one Input/Output Subsystem (IO) CPU, accelerated by up to 16 Compute Clusters (CCs), and we use asynchronous inter-cluster Remote Direct Memory Access (RDMA) transfers to exchange halo regions and use the main memory (external).

While OpenCL can also be used to deploy kernels onto the 16 CCs, this standard does not support local memory sharing between kernels. Indeed, all `__global` data are committed back to the main memory, and `__local` data does not persist between kernels. Such optimization feature makes kernel fusion optimization impossible; therefore, the main memory bandwidth becomes the main performance bottleneck. Vendor-specific extensions could be used to reuse memory between OpenCL kernels as in [THMB15], but these are non-standard and not part of the Kalray OpenCL offer. Moreover, the Kalray OpenCL host runtime requires Linux which cannot be used for efficient, soft real-time systems because of process scheduling jitter and system call overhead.

Halide [VZT+18], is a programming language (close to OpenVX) designed to exploit modern processors efficiently for tensor and computer vision applications. The Halide frond-end implements hardware independent computations like a parser, a scheduler, and an optimizer (memory allocation, synchronization, distribution). The execution part of our new distributed OpenVX framework could be a back-end for Halide to target the MPPA® processors.

9.1.3 OpenVX and OpenCL

OpenVX and OpenCL have a different role. OpenVX implements predefined kernels for computer vision and neural network applications, whereas OpenCL does not. Some OpenVX implementations are built on top of OpenCL. Indeed as mentioned above, the OpenVX implementation of AMD uses OpenCL to offload onto GPUs the computation of OpenVX kernels. Therefore, OpenCL is used at a lower level of implementation.

Moreover, as mentioned in Section 3.2.2, OpenCL exploits the memory hierarchy of the machine with the keywords `__global`, `__local` and `__private`, making OpenCL software challenging to write, and architecture dependent for the optimizations.

OpenVX hides the machine complexity from the programmer. Indeed, the OpenVX can be seen as an optimized library (like a DSL) for executing a Directed Acyclic Graph (DAG) of predefined kernels (functions). The standard also let the user add his/her own kernels. However, custom kernels are much more challenging to integrate and optimize automatically during the OpenVX graph optimization.

Finally, OpenCL defines the control-flow during the execution, whereas in OpenVX, the control-flow (control-path) is derived from the graph that is compiled and executed.

The OpenVX standard separates the control-path (kept implicit) from the data-path, providing inherently much more effective when exploited and implemented by System-on-Chip (SoC) manufacturers.

9.2 A Low-Level Distributed Offloading Engine

The offloading of computations from a host to one or several accelerators is not a trivial task. In the case of the MPPA® processor, the accelerators are the 16 Compute Clusters (CCs). As mentioned in Section 9.1.1, OpenVX is built as an acceleration programming API where the user application runs on a host CPU, and the described application graph can be executed anywhere: GPU, FPGA, and/or custom accelerators. With the MPPA® Multiprocessor System-on-Chip (MPSoC), one IO is the host and the CCs run the compute-intensive parts of the application graph.

Therefore, an offloading engine has been designed. Its design is inspired from the OpenACC [WSTaM12] runtime back-end of GNU Compiler Collection (GCC). OpenACC is the base of another famous offloading standard programming API, called OpenMP 4.0. As we target efficient and light-weight embedded computing, the offloading from a Linux Operating System (OS) running on the Input/Output Subsystem (IO) have been excluded.

We first describe the architecture of this offloading engine in Section 9.2.1. We provide key feature to performance in Section 9.2.2, and we explain the relationship between the OpenVX framework built on top of it in Section 9.2.3.

9.2.1 Architecture of the Offloading Engine

The offloading engine is responsible for the deployment of self-synchronizing computations. On the IO, the offloading engine operates in user-space of the multi-threading runtime presented in Chapter 6. OpenMP multi-threading is used to parallelize the distributed kernels inside each CC. Figure 9.2 shows the offloading engine architecture on which the OpenVX distributed framework is built.

The architecture of the offloading engine lies on the RDMA Network on Chip (NoC) of the MPPA® processor. The main memory is the Double Data Rate (DDR) memory that is accessed using one-sided operations for the CC. The IO has direct access to this memory using *Load-Store*. The use of the main memory is mandatory as most of the time

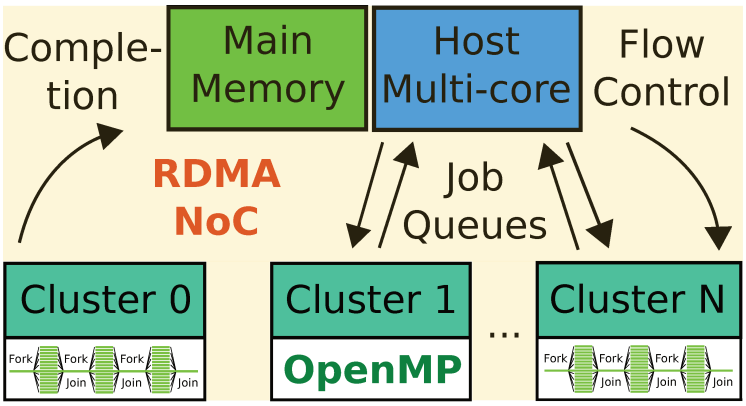


Figure 9.2 – *OpenVX Offloading Engine Architecture*

the on-chip memory (the sum of the array of local memories of MPPA®) is not big enough to accommodate all the buffers of a modern OpenVX application that uses full HD images for instance.

Load-Store does all local memory accesses performed by the cores inside the CC (intra-cluster). All remote memory accesses (inter-clusters and main memory) use RDMA for memory buffers and posted (Chapter 5) remote atomic operations for synchronizations. As seen in Chapter 4, local memory access is a low-latency memory transaction between a core and the internal cluster local memory (10-cycles latency for *Load/Store*), whereas a remote memory access uses the RDMA protocol over the NoC to access another CC memory or an off-chip memory (1200-cycles latency for *RDMA Put/Get*) with high throughput.

The OpenVX offloading engine implementation relies on the asynchronous one-sided communications and synchronization API over the NoC of the Kalray MPPA® processor, which provides high-throughput and low-latency RDMA, remote atomics and remote queue operations introduced in Chapter 5.

The offloading engine dispatches jobs and control commands using the job queues to the 16 CCs. Moreover, the identifier of CCs is virtual. It means that it is possible to subdivide pools of CCs without any changes to the running software inside the CC, as long as the offloaded kernels use global and local identifiers exposed by this offloading engine (similar to OpenCL).

9.2.2 Key Features of the Offloading Engine

The offloading engine provides the following set of features used by the OpenVX framework on the host side:

1) Multi-cluster platform topology creation. The platform topology is described using the OpenVX platform specific creation of the *context* as explained in Section 9.1.1. On MPPA®, the user describes the list of physical CCs used to offload the computation of the OpenVX graph, and the number of Processing Elements (PEs) that compute in parallel within the CC. Unused Processing Element (PE) are in idle mode.

2) Load / unload code byte stream to the CCs. The loading of the code from the main memory (external) to the local memory of the CC is performed by RDMA operations. We use a dynamic overlay to relocate the explicitly loaded code. The code has to be

compiled using the Position Independent Code (PIC) compile flag, and embedded in the host program (in the Input/Output Subsystem (IO)), so that the accelerators, namely the CCs can access the object file. The dynamic overlay is a major feature for time-predictable memory access of the CPU to the *.text* section. Indeed, the global system caches for accessing the instructions are not time-predictable and very difficult to analyze. Therefore, using a dynamic overlay is important in recent OpenVX Safety-Critical specifications (real-time performance and deterministic) presented in [Gid17].

3) Local buffer allocation associated to an identifier. The pre-allocation of buffers makes it possible to have an array of, off-line allocated, static pointers with associated identifiers. These memory areas are used to store the statically allocated data, that is performed by the distributed memory allocator presented in Section 9.3.3.

4) Execute kernel with arguments (name and arguments). The execution of kernels consists in sending the name of the function to be run, and its arguments. All sent jobs and commands are executed in order, by the selected CCs of the created execution platform. The execution of a kernel is always asynchronous for the host. The completion of the running kernels is provided to the host using a computation pipeline barrier. The barrier waits for the completion of outstanding kernels, running in the selected Compute Clusters (CCs).

5) Multi-cluster synchronization, synchronous or asynchronous collective. Asynchronous collectives regarding the host make the synchronization of a pool of Compute Clusters (CCs) within the pipeline possible, without any intervention of the host. It provides efficient synchronization mechanisms, controlled by the host CPU, to deal with multi-cluster Read-After-Write (RAW) dependencies in the main memory (external).

All of the above features are available through new primitives that have been designed for our needs. They are all executed by the host multi-core CPU, asynchronously to avoid stalls, and atomically to prevent data races. However, sent jobs and commands are processed in the execution order on the CCs side. Thus, pipeline barriers are provided to ensure the completion of all outstanding jobs and commands that were dispatched to the targeted CCs. In this way, transactions are always pipelined in job queues for execution efficiency with regards to the host. The offloading engine provides software flow-control mechanisms for the job queues to prevent data corruption when the multi-cluster system is congested.

The implementation does not implement locking mechanisms in the data path. Efficient software runtime implements separated control and data path. Functions like code relocation, pre-booting of OpenMP thread teams, memory allocations, and system initializations are performed once when the host starts the application. In the data path, the application uses pre-computed routes, a pre-loaded piece of code, and statically allocated memory buffers that make high-performance implementation possible.

As a result, at 500 MHz, the measured Input/Output Operation per Second (IOPS) from the host point of view is **731.3 kilo IOPS**, meaning an asynchronous request to a CC takes **681 machine cycles** on average. Such a throughput is enough for our OpenVX acceleration framework, built on top of this new offloading engine.

9.2.3 Integration and Usage in the OpenVX Framework

The OpenVX application runs on the host multi-core CPU and uses an acceleration API. The OpenVX *context* references the number of CCs in range $[0, 15]$ and the number of PEs in range $[0, 15]$ inside each compute cluster of an MPPA® processor. For the MPPA®, we propose an example of platform specific implementation in Figure 9.3.

```
struct _vx_platform platform = {
        /* use 8 clusters to process the graph starting from cluster 2 */
        .cluster_id_list = { 2, 3, 4, 5, 6, 7, 8, 9, 10 },
        /* use 8 clusters */
        .nb_cluster = 8,
        /* 16 PEs per cluster used by the OpenMP runtime */
        .nb_pe_per_cluster = 16,
        /* output OpenVX graph for debug */
        .dump_dotty_graph = 1,
        /* need kernel fusion for performance */
        .disable_kernel_fusion_optimization = 0,
};
...
/* build the context with requested MPPA specific OpenVX platform */
vx_context context = vxCreateContextFromPlatform(&platform);
```

Figure 9.3 – *Example of the Support Platform Description of the MPPA® Processor*

Each OpenVX node is distributed on **all** available CCs (flat distribution) linked to the OpenVX *context*. The distribution is achieved by operating a low-level kernel offloading engine in a lightweight multi-threaded runtime onto the host CPU.

The parallelization relies on OpenMP3 `# pragma omp parallel` for work sharing between cores inside a CC and uses the RDMA NoC API [HdDdMH17] to perform inter-cluster data transfers and main memory accesses.

9.3 Online Optimizations: *vxVerifyGraph*

The framework for running OpenVX applications on stand-alone clustered manycore processors is based on a distributed runtime execution environment. Starting from [WM14], which targets *Load/Store* CPU+GPU architectures with shared memory, we adapt and automate optimizations for both *Load/Store* (synchronous, intra-cluster) and RDMA (asynchronous, inter-cluster) types of memory accesses.

9.3.1 Optimization Workflow

The workflow specifies the automatic steps performed during the OpenVX graph dynamic optimization *vxVerifyGraph*. The workflow is executed onto the embedded host; thus, graph optimization can be done at runtime if external parameters change. As shown in Figure 9.4, the workflow takes as input the OpenVX application and produces computation commands for one or several accelerators.

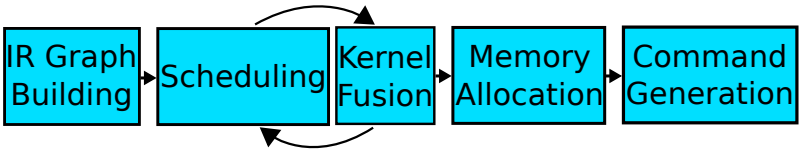


Figure 9.4 – *OpenVX Verify Graph Workflow - vxVerifyGraph [G+ 17]*

a) IR Graph Building provides the internal Intermediate Representation (IR), a Single-Rate (SR) Directed Acyclic Graph (DAG) on which the next passes of the optimization workflow operate. The graph builder takes user buffers which are OpenVX objects, looks for adjacent nodes using a Depth-First Search (DFS) and propagates object properties to buffers and nodes, such as image sizes and configuration parameters of OpenVX kernels.

Several errors may be detected and dealt with during the graph building process: unconnected buffers or nodes, cycles, multiple buffer writers, and the absence of input or output buffers for the OpenVX application. When errors are detected, the graph building results in failure giving the user the list of implicated nodes or buffers.

b) Scheduling is based on a topological sort of the Single-Rate (SR)-DAG presented in [KL95]. It is performed to enforce the graph dependencies for kernel executions, and its complexity is $O(n)$. In practice, we implemented the Algorithm 1 presented in Section 3.3.

c) Kernel Fusion is the critical optimization that let kernels reuse data already copied in the on-chip memory (local memories). This optimization aims at reducing the main memory (external) bandwidth. Adjacent kernels (nodes) are fused to make this optimization possible. Kernel fusion opportunities are identified by a simple constraint satisfaction algorithm that ensures memory allocation feasibility. The schedule is updated after each kernel fusion. The complexity of the fusion optimization algorithm is $O(n)$ where n is the number of kernels. The kernel fusion optimization is presented in details in Section 9.3.2.

d) Memory Allocation pass is performed by an allocator of distributed memory operating on the schedule (once the fusion optimization is completed). As explained in 9.3.3, virtual buffers are allocated in the main memory (external) or the internal on-chip memories (local memories).

e) Command Generation performs the computation of arguments for the RDMA-based tiling engine. The commands are saved in lookup tables. The runtime of the RDMA-based tiling engine running the compute clusters is presented in 9.4. The basic tiling principle is to split a buffer (1D or 2D) into tiles and to distribute them onto the computing resources. Once commands are generated, the *vxProcessGraph* consists in sending commands to the CCs as seen in Section 9.2. The commands are sent asynchronously, but their executions are scheduled in order across the matrix of CCs.

9.3.2 Automatic Kernel Fusion Optimizations

As explained in Chapter 7, the kernel fusion optimization consists in grouping two adjacent kernels together to avoid temporary buffers being copied to the external memory. Again, this technique is inspired by the *Pairwise Grouping of Adjacent Nodes* algorithm [BML99].

However, in this chapter, the grouping or kernel fusion algorithm is different. The **kernel fusion optimization operates at a multi-cluster level as each kernel is distributed on the entire CC matrix** to achieve low latency.

As each vertex of the SR-DAG is distributed on all available CCs, it makes data dependency between fused kernels a multi-dimensional problem as shown in Figure 9.7. Fusion decisions are based on the following constraints: the pattern type of kernel to fuse, the

amount of local memory required, and the type of input and/or output buffers have to be virtual (see Section 9.1.1). The real buffers cannot be fused because in the OpenVX specification, these buffers should stay accessible by the host application. The $O(n)$ fusion optimization pass, takes the main graph schedule as an input and produces a new schedule that represents the new fused kernels.

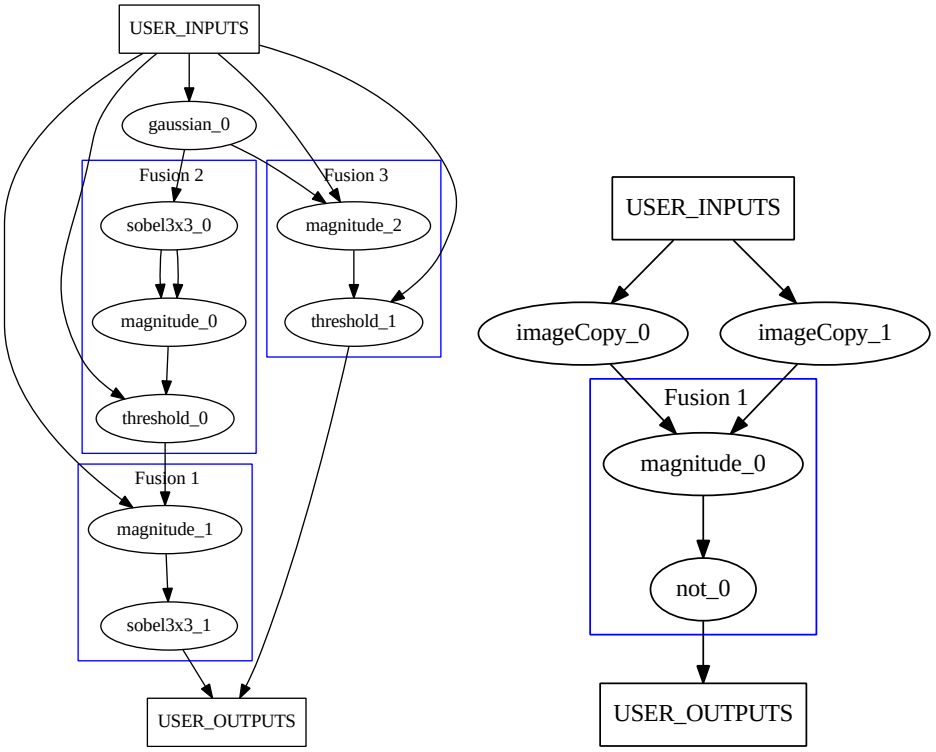


Figure 9.5 – *Example of a Graph Display from the IO, Schedule and Fusion Optimizations*

This new schedule is then placed in the main graph schedule until all fusion opportunities are applied to the application graph. The scheduling policy consists of executing fused kernels in depth first. The supported patterns of kernel fusion are any combinations of point-to-point operator kernels using overlap tiling or not.

The fusion optimization avoids recomputing halo regions and removes useless memory copies for the management of halo regions. However, it is involved regarding inter-cluster data transfers and the memory allocation of input and output tiles, as buffers need to be padded on the borders for halo exchange (see borders of distributed tiles in Figure 9.7).

Moreover, for the debugging of the scheduling optimizations of the graph, we have implemented a stubbed function over PCIE to visualize the OpenVX computation graph defined by the user. If not disabled by the platform description already presented and shown in Figure 9.3. It allows the user to check the described graph and successful optimization, as well as, the schedule of the nodes.

Figure 9.5 shows an example of an OpenVX graph schedule, dumped by the host. The *USER_INPUTS* boxes are the input buffers and the *USER_OUTPUTS* boxes are the output buffers of each graph. These buffers are *real* buffers that can be accessed by the host application. Kernel fusion decisions are shown by a blue box over the kernels in Figure 9.5. On the left graph, three fusion optimizations have been automatically found, and on the right graph, one fusion optimization has been performed.

9.3.3 Distributed Static Memory Allocation

The distributed memory allocator manages the memory consumed by the virtual buffers of the OpenVX application. User buffers are already allocated at object creation (buffers). The distributed memory allocation operates after the scheduling and the kernel fusion passes. The allocator has two memory pools which are the array of symmetric local memories and the main memory. The memory allocation is mainly governed by the graph schedule, through the lifetime of virtual objects, the kernel fusion decisions, the kernel dependency patterns, spills in the main memory for user buffers, and also the N-buffering and tiling configurations parameters usually depending on image sizes.

By default, the runtime automatically spills buffers to the main memory (external) during the computation when the local memories are full.

The RDMA-based tiler, described in Algorithm 12, spills and tiles images that do not fit into the available local memories. Inside each compute cluster, a memory area of 1.4 megabytes is reserved at the OpenVX context creation. This memory buffer size is configured in the OpenVX platform's specific files of the framework itself but is easily tunable to target any RDMA-enabled clustered manycores.

This memory area contains temporary multidimensional buffers (virtual buffers of the OpenVX standard) that are allocated by a first-fit memory allocator giving buffer offsets in the local memory of each CC. The first-fit algorithm takes buffers related to vertices in their schedule list order and recycles the memory once their live range has ended. On classical OpenVX applications, 4 buffers are allocated in each local memory before being reused.

Finally, the memory allocation is guaranteed to succeed as the kernel fusion optimization pass is aware of the available size remaining in the local memories when fusing kernels. Indeed, when the kernel fusion requires too much memory, the fusing optimization pass chooses the RDMA-based tiler to spill on the main memory. The RDMA-based tiler splits the computation automatically to make it fit in the local memories, thanks to Algorithm 12.

9.4 Explicit RDMA-based Communication Engines

9.4.1 A Tiling & Fusion RDMA Engine

The RDMA-based tiler operates at runtime (graph execution) inside each CC concurrently, **distributing the execution of each OpenVX node across the entire matrix of compute clusters**. This technique is essential to achieve low-latency execution moreover, is unlike classic dataflow graph execution, where actors are mapped to different CCs [dDAB+13, CDG+14]. Algorithm 12 receives commands through the job queues as seen in Figure 9.2 when the host application calls *vxProcessGraph*. Command arguments are the input and output images, tile geometries, halo geometries, the N-buffering configuration to absorb main memory latency; the start compute offset in images stored in the main memory (external) for each CC and the number of CCs that execute the kernel concurrently. Halo geometries are provided by a hand-written oracle, which is used during the scheduling.

Algorithm 12 An Automatic Distributed RDMA-based Overlap Tiler Concurrently Operating onto Multiple Compute Clusters.

```
1: Input: InImg, Width, Height, NbTotalTiles, N, TileWidth, TileHeight, HxIn/Out,
   HyIn/Out, NbTileStartOff, NbTiles
2: Output: OutImg
3: /* Set multidimensional pointers in local memory */
4: Set InTiles[N][TileHeight+2*HyIn][TileWidth+2*HxIn]
5: Set OutTiles[N][TileHeight+2*HyOut][TileWidth+2*HxOut]
6: for i := 0 to N-1 step 1 /* Warm up the pipeline */ do
7:    InTilesEvent[i] ← Asynchronous Get Stride-to-Dense from (In-
      Img+NbTileStartOff+i) to InTiles[i]
8: end for
9: for i := N to NbTiles+N step 1 /*Pipeline Loop */ do
10:   ProcIdx := (i-N)%N /* Compute Buffer Index */
11:   FetchIdx := i%N /* Prefetch Buffer Index */
12:   /* Wait for DMA Transactions Completions */
13:   Wait Get InTilesEvent[ProcIdx]
14:   /* Only one wait if in-place computation */
15:   Wait Put OutTilesEvent[ProcIdx]
16:   /* Compute Tile i-N in Parallel in the Node */
17:   OutTile[ProcIdx] := Kernel(InTiles[ProcIdx])
18:   if OutImg is local then
19:      Async. Puts of halo regions to adjacent compute clusters for fused kernels depen-
         dencies
20:   else
21:      if i < NbTiles+N then
22:         OutTiles[ProcIdx] ← Async. Put Dense-to-Stride to (Out-
            Img+NbTileStartOff+i) from OutTiles[ProcIdx] /* Write to Main Memory
            */
23:      end if
24:   end if
25:   /* Prefetch Tile i from Main Memory */
26:   if i < NbTiles then
27:      InTilesEvent[FetchIdx] ← Async. Get Stride-to-Dense from (In-
         Img+NbTileStartOff+i) to InTiles[FetchIdx]
28:   end if
29: end for
30: Async. Fence /* Memory Consistency, Mandatory for Global Read-After-Write Depen-
   dencies */
31: Synchronize NbNodes Clusters /* Ordered with Fences */
```

Firstly, the distributed tiler either retrieves input tiles from the main memory using N-buffering (Line 7) and sets local multidimensional input pointers (Lines 4 and 5) to previous local output buffers of a previously executed kernel when it is "fused" with the current one.

Secondly, the master thread of the CC calls the compute kernel (Line 17). It performs intra-cluster parallelization with OpenMP compilation directives.

Thirdly, the output is either copied back to the main memory for OpenVX user buffers (Line 22) or remains local, if the next kernel is fused with the current one. When the

next kernel is fused, depending on kernel fusion patterns, halo exchanges are initiated to adjacent compute clusters to satisfy inter-cluster data dependencies (Line 19).

Finally, memory consistency operations are initiated to memories that have outstanding writes (Line 30) before the multi-cluster synchronization (Line 31).

9.4.2 Tiling & Fusion Optimizations

This section illustrates the multi-cluster tiling, and the tiling combined with the fusion optimization at the multi-cluster level.

Tiling

Figure 9.6 shows the tiling distribution for a 2D stencil computation using 4 CCs. A typical use case would be an edge detector followed by morphological operators. Steps 1, 3, and 5 a copy data from/to the main memory from/to the local memories of the CCs, which is shown in Algorithm 12 at Lines 7 and 22. Red arrows show main memory (external) spaced [HdDdMH17] transfers with the local memories (on-chip memory). The Algorithm 12 performs automatic data prefetching to hide the main memory access latency. As explained in [WWP09], reducing latency by software and hardware prefetching is a key to performance. Memory accesses are often the bottleneck in high-performance computing. Steps 2 and 4 perform the computations in parallel in each used CCs.

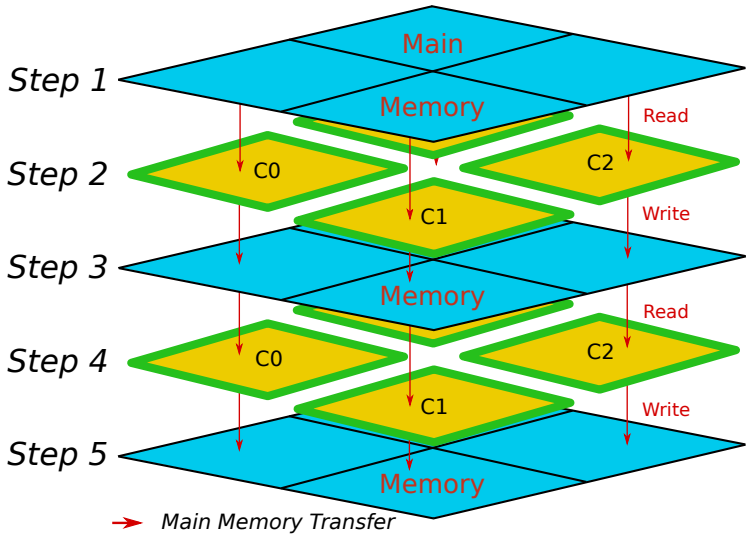


Figure 9.6 – *Automated Multi-clusters Tiling*

Tiling can be improved as the main memory bandwidth wall is a bottleneck. The fusion optimization is proposed here to tackle this problem.

Tiling Combined with Fusion

Figure 9.7 shows the tiling combined with kernel fusion optimization, also using a 2D stencil computation with 4 CCs. Steps 1 and 5 perform copy from/to the main memory. At step 3, the black arrows represent strided inter-cluster asynchronous RDMA transfers, also shown Line 27 of Algorithm 12.

When fused kernels are executed, each CC stores tiles that are automatically reused from one kernel to the next. Therefore, the N-buffering N variable of Algorithm 12 is always

set to 1 when fusing to maximize the on-chip memory usage, minimize data transfers, and save main memory bandwidth. We exploit inter-cluster RDMA data transfers to reduce the number of external memory accesses.

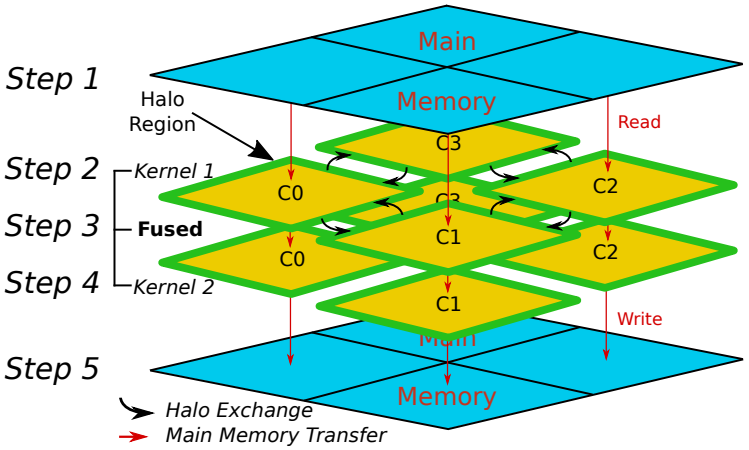


Figure 9.7 – *Automated Multi-clusters Tiling Combined with Fusion*

Thanks to the fusion optimization, we gain a factor of 2 on the data copied in the main memory. Indeed, the main memory bandwidth (external) is often a performance bottleneck for manycore architectures [WWP09].

The RDMA-based tiler (Algorithm 12) can be used straight out of the box by other architectures supporting asynchronous one-sided communications, such as OpenCL `async_work_group_copy()`, Message Passing Interface (MPI)-3 one-sided operations, or even the low-level onto the eDMA feature of the Texas Instruments (TI) Keystone II.

9.5 Complex Distribution and Memory Access Patterns

DMA-enabled architectures have always been challenging to program, mainly because all data transfers are explicit.

9.5.1 Dealing with Irregular Memory Accesses

Irregular memory accesses happen in some OpenVX standard kernels like geometrical transformations. These irregular memory access patterns are difficult to predict during the verification and optimization of the graph in *vxVerifyGraph*. Therefore, the proposed tiling solutions presented in Section 9.4 cannot be used for such problems. There exist two types of geometrical transformations which are the affine and the perspective transformations in OpenVX.

The warp affine transformation is defined by $2*3$ matrix. It translates pixel coordinates from the input to the output image. Let us consider M the transformation matrix, O and I respectively the output and input image. We use the formula provided as follows and presented in the *Warp Affine* section of [G+17]:

$$xt = M_{1,1} * x + M_{1,2} * y + M_{1,3}$$
$$yt = M_{2,1} * x + M_{2,2} * y + M_{2,3}$$
$$O(x, y) = I(xt, yt)$$

The warp perspective transformation is defined with a $3 * 3$ matrix. It translates and rotates pixel coordinates from the input to the output image. Let us consider M the transformation matrix, O and I respectively the output and input image. We use the formula provided as follows and presented in the *Warp Perspective* section of [G+17]:

$$xt = M_{1,1} * x + M_{1,2} * y + M_{1,3}$$
$$yt = M_{2,1} * x + M_{2,2} * y + M_{2,3}$$
$$zt = M_{3,1} * x + M_{3,2} * y + M_{3,3}$$
$$O(x, y) = I(\frac{xt}{zt}, \frac{yt}{zt})$$

To compute the output O, we iterate on the output pixel coordinates x and y. We compute the coordinates xt, yt and zt of the input pixels I in the following cases:

- Warp Affine: $O(x, y) = I(xt, yt)$
- Warp Perspective: $O(x, y) = I(\frac{xt}{zt}, \frac{yt}{zt})$

The matrix M is provided by the OpenVX user, thus **unknown**, meaning that memory access patterns can be irregular.

The computation of the coordinates is performed using floating point operations and converted back to integers once computed. The computed coordinates are used to access the input pixel and write this accessed input pixel on the (x, y) coordinates of the output image.

We show an example of a geometrical transformation in Figure 9.8. The transformation is a rotation using the warp perspective. When there is no correspondence between the output pixel and the input pixel, a black pixel is set.

Figure 9.8 – *Example of Geometrical Transformation, namely a Rotation.*

Irregular memory patterns are challenging to manage on DMA-enabled architectures. As such when the data movements depend on the input data itself, it is hard to handle efficiently. Such a problem usually requires a global cache system. Kalray provides at low-level an implementation of the Distributed Shared Memory (DSM) system where the PEs of the CCs matrix have direct access to the main memory using the Memory Management Unit (MMU) that is presented in Section 2.4.5. However, the current DSM system is not acceptable in our proposed OpenVX framework because of the following issues:

- It requires at least 500 kilobytes of local memory to operate, and the local memory is a critical resource for performance.
- It cannot be unplugged dynamically when a kernel of a graph does not need it.
- It does not implement prefetching mechanisms.

Our contribution to the efficient execution of these OpenVX kernels is to implement a cache of 2D tiles in Algorithm 13. The tile geometry is set with the *TileWidth* and *TileHeight* parameter at Line 1, at initialization time. Furthermore, arguments of this algorithm define the geometry of the 2D tiles that are cached with parameters *CacheWidth* and *CacheHeight*. The goal of such a system is to fetch on demand and depending on the transformation, **the input pixel to be placed in the output pixel**. The cache of tiles is dedicated to the geometrical transformations, but the idea can be applied to other use-cases.

Such configurability let the graph verification pass of OpenVX analyze the geometrical transformations and configure the cache accordingly to get a high rate of hits in fetched tiles.

Algorithm 13 runs on each PE of each CC running the geometrical transformation concurrently. As in Section 9.4, the output image is tiled inside the CC, and each PE operates on an image tile. The parallelization of a sub-tile inside one CC is managed by OpenMP. Moreover, the loop presented in Line 7 of Algorithm 13 is tiled to provide a better locality of 2D tiling at the global level of the explicit cache of tiles.

Black pixels in Figure 9.8 (top of the right image) corresponds to computed input coordinates that were out of the input image geometry. The condition observed in Line 9 of Algorithm 13 manages this corner case. Initialization of the output image to black could be done at initialization time; but, for performance, it is performed while iterating on the output pixels, on-the-fly. This optimization avoids accessing the entire output tile at initialization time, thus reducing memory accesses.

With such an algorithm architecture, fined tuning could be done during the verify graph process to configure the cache with the following parameters: line width and height, tile offset of in the image (middle, end, beginning, custom). We tuned our new cache of 2D tiles to fetch in lines for x translations and y translations for affine transformations, but further prospection and effort could be done for perspective transformations.

9.5.2 Implementation of Distributed Reduction and Dynamic List Update

The Harris corner detection is part of the OpenVX standard, and it is used to detect key points within an image. For instance, these points can later be used for motion vector estimation.

The Harris corner detection algorithm requires the implementation of reductions when running in parallel. Reductions are explained in Section 3.1.3. In parallel computing, when we consider a distributed machine with local memories like the Kalray MPPA®, the parallel implementation of reduction is not easy. The Asynchronous One-Sided (AOS) library (Chapter 5) has been developed to solve this problem, thanks to remote atomic operations. Indeed remote atomic operations are powerful to deal with atomic updates of variables in distributed memory systems fitted with local memories (only).

The Harris corner detection performs a reduction at some points during the execution to compute the maximum value of an image. This image is split across the array of local

Algorithm 13 A Reconfigurable 2D Explicit Cache of Tiles for Geometrical Transformations. Each PE of a CC implements its own 2D cache.

```
1: Input: InImg, MatrixTransformation, TileWidth, TileHeight, NbTiles, CacheWidth,
   CacheHeight
2: Output: OutImg
3: /* Set multidimensional pointers in local memory */
4: Set InTile[CacheHeight][CacheWidth] /* Each PE has it own */
5: Set OutTile[TileHeight][TileWidth] /* Each PE has it own */
6: for i := 0 to NbTiles step 1 /*Pipeline Loop */ do
7:    for OutPoint in OutTile /*Loop over pixels in parallel */ do
8:       Compute InPoint = MatrixTransformation * OutPoint
9:       if InPoint is out of bound of InImg then
10:         Write Border Pixel in OutPoint
11:      else
12:         if Check InPoint Hit in Current InTile then
13:            Write InPoint in OutPoint
14:         else
15:            Async. Get of a new InTile from InImg
16:            Write InPoint in OutPoint
17:         end if
18:      end if
19:   end for
20:   Async. Put of Current OutTile in OutImg
21: end for
22: Async. Fence /* Memory Consistency, Mandatory for Global Read-After-Write Depen-
    dencies */
23: Synchronize NbNodes Clusters /* Ordered with Fences */
```

memories of different CCs. The image tiling is the same as in Figure 9.6, but for one kernel only. Our implementation is derived from the classical MapReduce programming model used in the parallel multi-cluster implementation.

In the Harris corner detection algorithm, a maximum value is computed. As this computation is spread on all the available CCs, each CC computes its own local maximum value and sends it to all other CCs computing the distributed Harris corner detection node. Therefore, an all-to-all CCs data communication is performed to send all CCs the contribution of each CC.

We use asynchronous RDMA *Put* operations followed by posted remote atomic (*postadd*) operations to unlock all remote CCs. Thus, after posting these operations, each CC waits on the variable to be unlocked. As RDMA operations are ordered with posted remote atomic operations, when the CC is unlocked, it immediately performs the local computing of the maximum value of each contributor (the other CCs).

Finally, the Harris corner detector outputs a list of coordinates of points that need to be written back in the main memory (external), to be directly accessible by the host, hosting the OpenVX user application. Each CC has to write its local list of found coordinates. The list sizes are different, and they depend on the image content. Therefore, we use the remote atomic named *fetchadd* (see Chapter 5) on a counter mapped in the main memory. This counter is initialized to 0 at the beginning of the execution by the host. The fetched value in the CC is then used to write at a proper offset in the list (OpenVX vx_array

object) mapped in the main memory (external) as well. Thanks to this algorithm, it is possible to combine efficiently the contribution of several CCs.

9.6 Results & Discussions

Optimizations performed by the framework are fully automated. They do not require any user inputs. This section shows the impact of automatic optimization passes regarding fusion and prefetching on the execution time. The graph verification and scheduling were done offline for benchmarking. The entire distributed framework (workflow, runtime, and kernels) has been implemented in standard C99 for efficient execution in embedded systems, and without any complex library dependency but the C library.

Before explaining the results, we highlight three cases of speedups:

- **Sub-linear**: means that the parallelization is below the theoretical speedup when increasing the number of cores. Such a low speedup is due to several known parallel programming problems, which are the Amdhal's law (some part of the application cannot be run in parallel), the overhead of the extra software control for the parallelization, and the data transfers used to spread data across the available cores.
- **Linear**: means that the cores are well exploited, and the speedup increases linearly with the number of cores. The application is embarrassingly parallel usually when the computations are independent (no data dependency between cores and the NoC communications are not stressed).
- **Super-linear**: means that the speedup factors are above the number of cores. Such a speedup is usually misunderstood as it somehow breaks classical speedup laws that bound the maximum theoretical speedup of parallel applications, like the Amdhal's law. However, super-linear speedups can be obtained in a few cases, usually in HPC. It can be observed while focusing optimizations onto memory accesses [WWP09], while eliminating costly stalls on the main memory accesses, and exploiting **shared** on-chip memories very efficiently.

9.6.1 Performances Analysis

We use single-channel images (VX_DF_IMAGE_U8) for benchmarking with image sizes corresponding to VGA (480p) and full HD (1080p). Strong scaling is shown when varying the number of CCs, and the number of PEs is set to 16 within each CC. The operating chip frequency is 500 MHz. It uses a single DDR3 channel running at 1333 MHz. The power consumption of the chip varies from 4 to 12 Watts, depending on the use case and the optimizations applied (fusion, prefetching and core-level optimization). We use point operator kernels using tiling or overlap tiling techniques with either halo regions inter-cluster data transfers or spilling, depending on the optimization level.

9.6.2 Benefits of Asynchronous RDMA Prefetching

Figures 9.9 and 9.10 compare the execution latency of a single kernel using synchronous strided-to-dense RDMA main memory accesses compared to asynchronous accesses implementing N-buffering (see *bench*_N_BUFF results). Five kernels (*copy, conv3x3, threshold, dilate, or*) have been evaluated. We found that asynchronous RDMA prefetch is a must-have for performance, as long as the main memory is not the bottleneck. Our RDMA

prefetching mechanism provides up to 80% better performance than the blocking memory accesses. Quasi-linear speedups are obtained for up to 8 clusters before becoming memory bound with the main memory (external memory).

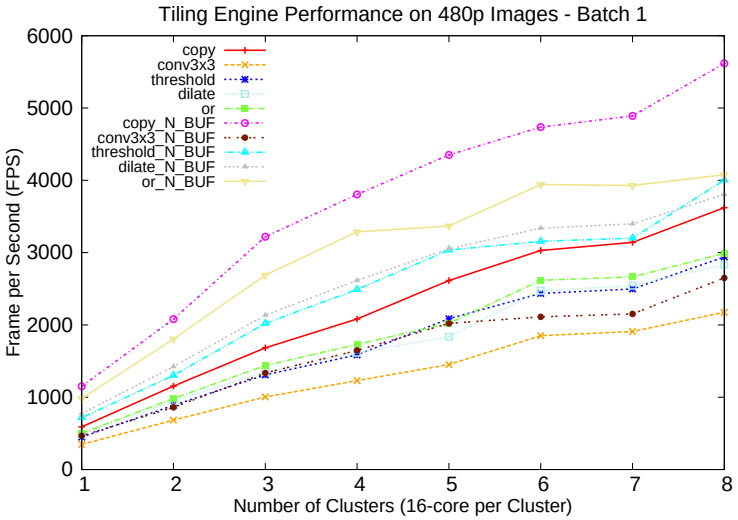


Figure 9.9 – *Automatic Tiling Engine Performance. VGA Images. Simple Tiling vs Tiling with N-Buffering (N_BUF = N-Buffering = Prefetch).*

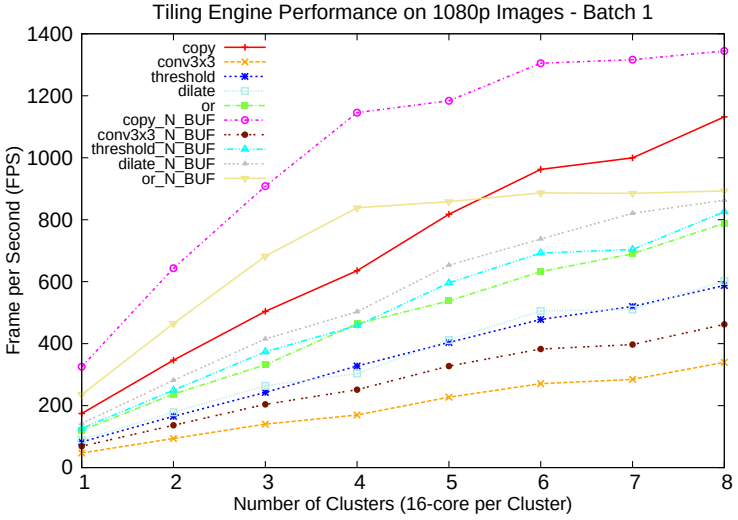


Figure 9.10 – *Automatic Tiling Engine Performance. Full HD Images. Simple Tiling vs Tiling with N-Buffering (N_BUF = N-Buffering = Prefetch).*

9.6.3 Automatic Kernel Fusion

Figures 9.11 and 9.12 compare results with tiling with prefetch, and tiling with prefetch combined with kernel fusion. As seen in Figure 9.12 for full HD images with the *edge_detect_pipeline* (Median, Sharr and Magnitude pipeline), the kernel fusion optimizer can fuse the kernels when the execution platform integrates 10 CCs and more.

Indeed, with this *edge_detect_pipeline*, the entire data set fills all available local memories, when the number of CCs is higher than 10, making the fusion optimization possible

(Figure 9.7). Similar speedups are also noticed for other cases for both image resolutions (full HD and VGA).

Data copies in the main memory (external) are avoided, thus providing an extra speedup of 25 % in this case compared to the spilling N-buffering version.

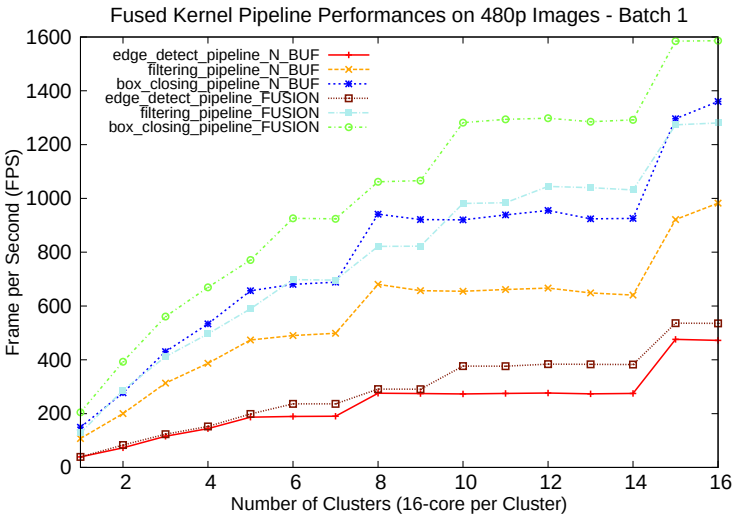


Figure 9.11 – *Automatic RDMA-based Kernel Fusion Performance. VGA Images. Tiling with N-Buffering (N_BUF = N-Buffering = Prefetch) vs Kernel Fusing (FUSION).*

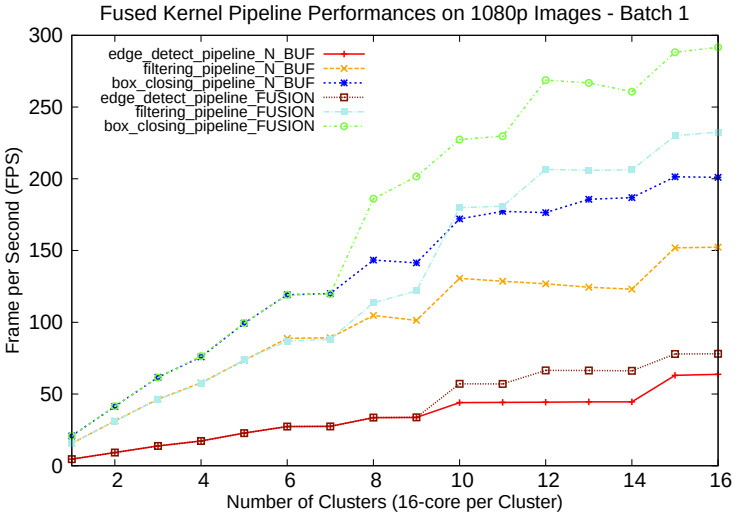


Figure 9.12 – *Automatic RDMA-based Kernel Fusion Performance. Full HD Images. Tiling with N-Buffering (N_BUF = N-Buffering = Prefetch) vs Kernel Fusing (FUSION).*

9.6.4 Super-linear Speedup at Multi-Cluster Level

In the *edge_detect_pipeline* of Figure 9.12, super-linear speedups are observed. On full HD images (1080p), 1 compute cluster provides 4.62 Frames per second (fps) and the 16-cluster version with kernel fusions, asynchronous strided inter-cluster halo regions exchange provides 78.07 fps, for a speedup of 16.9. As already seen, super-linear speedups are usually misunderstood as they contradict the classical theoretical speedup law's. On complex memory hierarchy processors, super-linear speedups are achieved by optimizing memory

accesses at multiple levels of the memory hierarchy. Several parameters need to be taken into account. These parameters are memory access locality (shared cache or local shared-memory), multiple levels of tiling geometries, and asynchronous prefetching mechanisms. On the Kalray MPPA® processor, such speedup is obtained thanks to the exploitation of the on-chip local memories, and the use of asynchronous (strided) inter-cluster data transfers, which eliminate main memory access stalls.

The main memory (external) bandwidth wall needs to be avoided to fully exploit the processing capabilities of low-power massively parallel architectures [WWP09]. Other cases, *filtering_pipeline* (Sobel, Magnitude, Erode & Dilate) and *box_closing_pipeline* (Conv3x3, Erode & Threshold) have both speedups of 15.1 on full HD images.

9.6.5 Irregular Memory Accesses Performance

With this contribution, we let geometrical transformation algorithms run on the targeted clustered manycore processor. As already explained, irregular access patterns are difficult to manage on clustered architectures. In Figures 9.13 and 9.14, we show the multi-clusters throughput (fps) for geometrical transformations. In the CC, all PEs run in parallel the 2D explicit cache of tiles. Although we observe at best a speedup of 10 with full HD images for the translation, the performance for the Warp Perspective transformation is poor. Indeed the Warp Affine transformation (translation) still has more regular memory access patterns (better data locality).

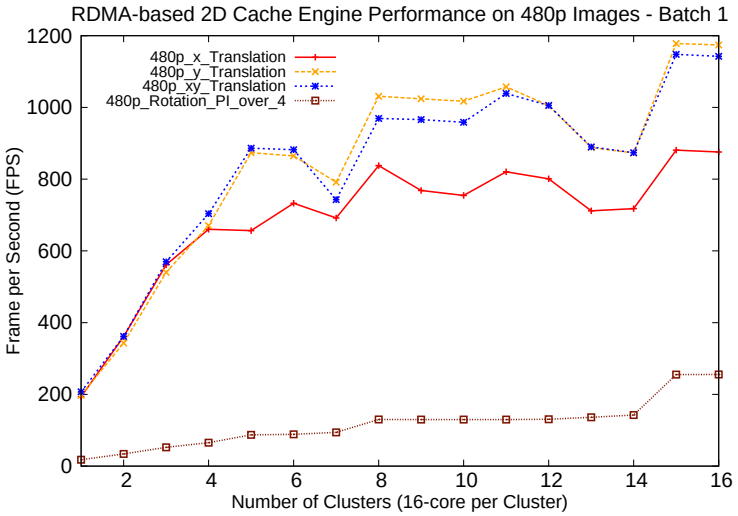


Figure 9.13 – *RDMA-based 2D Explicit Cache of Tiles Performance. VGA Images.*

More advanced static analysis during the *vxGraphVerify* process could be done to configure the 2D explicit cache of tiles better. Still, the mechanism is generic and always working, only fine-tuning is now required to carefully set the input parameters of to 2D explicit cache of tiles (offset, 2D dimension, local tiling). Such optimization is not the purpose of this thesis.

In a mono-cluster implementation, when the platform-specific attribute *nb_pe_per_cluster* is in the range $[1, 16]$, the performance also shows little speedups in Figures 9.15 and 9.16. The best-observed speedup is for the XY translation where we show a speedup of 14 between 1 and 16 PEs running the 2D explicit cache of tiles.

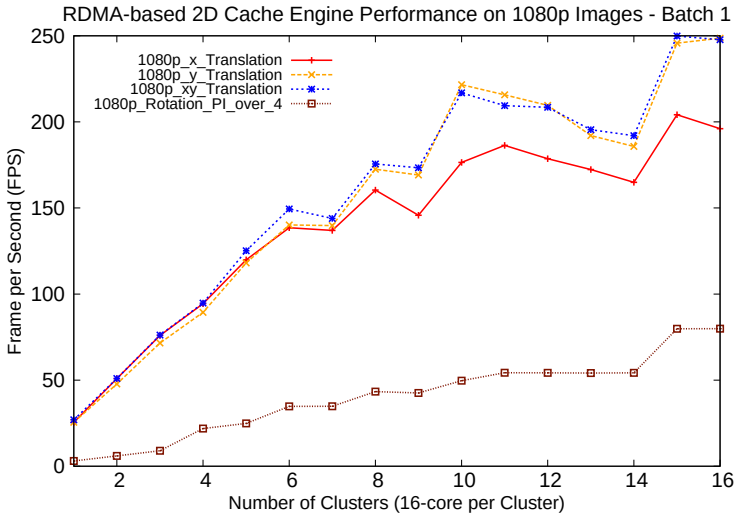


Figure 9.14 – *RDMA-based 2D Explicit Cache of Tiles Performance. Full HD Images.*

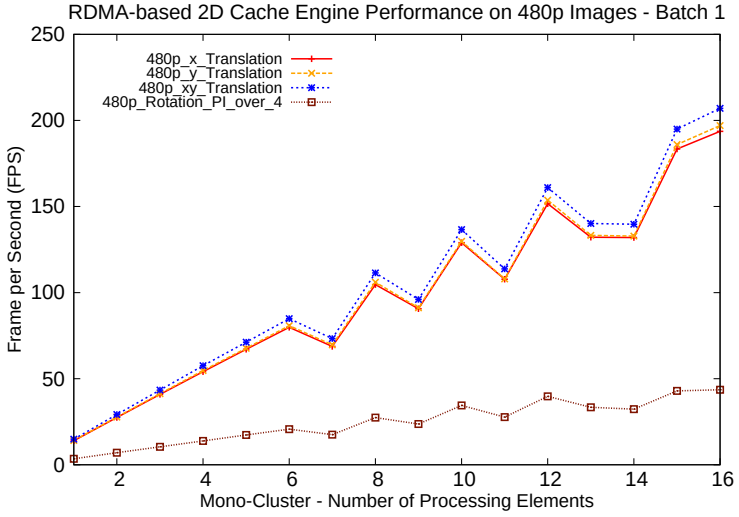


Figure 9.15 – *Mono-Cluster RDMA-based 2D Explicit Cache of Tiles Performance. VGA Images.*

9.6.6 Performance of the Harris Feature Point Detection

In Table 9.1, we show the multi-clusters performance for the detection of feature points within an image. The current implementation has limitations. It can be observed in Table 9.1 that all cluster combinations are not supported. There is currently a limitation as only the fusion optimization part has been implemented, and not the generic implementation that makes the execution of the Harris corner detector possible independently of the amount of memory available in multiple CCs. Such a feature should be implemented in the future.

Still, inside each CC, 16 PEs are processing the local tile concurrently. At the multi-cluster level we show a speedup of 5.6 between 2 CCs and 16 CCs of VGA images (theoretical is 8). A speedup of 1.85 is observed between 8 CCs and 16 CCs (theoretical is 2). No super-linear speedups are observed in this benchmark, as only the fusion optimization has been implemented, and also, as the algorithm requires reductions. Reductions are se-

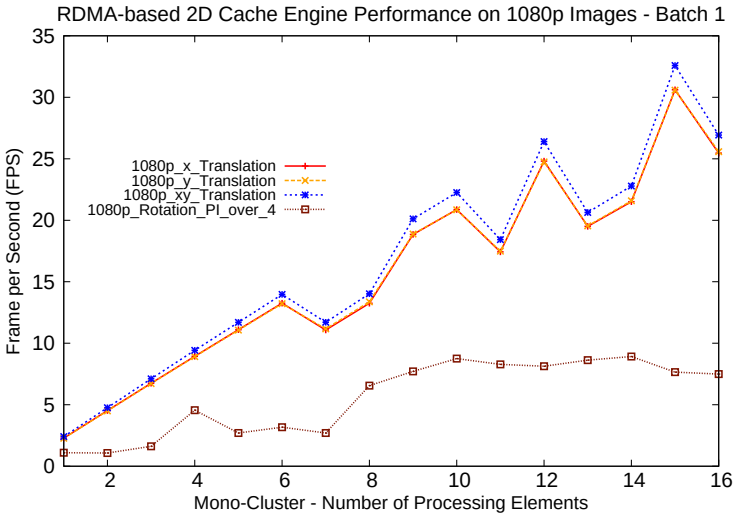


Figure 9.16 – *Mono-Cluster RDMA-based 2D Explicit Cache of Tiles Performance. Full HD Images.*

quential at the memory system point of view. Thus, it implies additional synchronizations with remote atomic operations. It is hard to get linear speedups with such constraints.

Benchmark Harris Corners Number of Clusters	Video VGA 640 x 480 (fps)	Video HD 1280 x 720 (fps)	Video Full HD 1920 x 1080 (fps)
2	136	-	-
4	247	-	-
6	336	124	-
8	454	162	-
10	484	190	-
12	575	230	-
14	575	231	-
16	770	295	124

Table 9.1 – *Multi-cluster Performance of the Harris Corner Detection of OpenVX on MPPA® in fps*

9.7 Conclusion

In this chapter, we describe the implementation and benchmark the experimental results of the first OpenVX framework for the Kalray MPPA® processor second generation. The framework is built for the low-latency execution of OpenVX application graphs. Implemented on top of the AOS API (see Chapter 5), and the new multi-threading runtime (see Chapter 6), the entire framework has been written from scratch in C, starting from the API specification provided by the Khronos group (around 20 thousand lines of code).

The main focus is put here on the throughput of the explicit data communications for enabling low-latency execution of an OpenVX graph. The strategy consists in mapping each OpenVX node of the graph on all available Compute Clusters (CCs) for the acceleration matrix of MPPA®. The parallelization of each OpenVX kernel (node) is fully automated from the point of view of the user of OpenVX.

On massively parallel architectures, one of the main performance bottlenecks is the main memory bandwidth (external memory). The main memory (external) also features long memory access latencies. Our framework automatically addresses these two bottlenecks by exploiting the on-chip local memories as much as possible (kernel fusion), and by operating RDMA engines in asynchronous mode (data prefetching).

The framework performs automated optimizations, including kernel fusion, kernel execution tiling, and N-buffering of external memory transfers. The kernel fusion technique eliminates intermediate buffers and main memory (external) accesses. Thus, it saves main memory bandwidth that is most of the time the performance bottleneck [WWP09].

One of the main contributions is the automatic distribution of OpenVX nodes on the available local memories of the CCs. Thanks to such a feature, we can reach low-latency execution. However, this strategy requires asynchronous inter-cluster RDMA transfers to handle multi-cluster RAW dependencies, which are complicated to implement.

The results are measured using the real MPPA® hardware. We show some linear and super-linear speedups at the multi-cluster level, which demonstrate that the MPSoC is well exploited.

Moreover, another significant contribution in this Chapter is the design of an offloading engine presented in Section 9.2. This engine offers a very efficient entry point from the host to offload and control the multi-core CPUs (Compute Clusters (CCs)) of the MPPA® MPSoC. Thanks to this contribution, the usefulness and the ease of use of this engine, it is today the back-end of several optimized libraries offloaded from the IO to the CCs. These libraries are the BLIS [VZVDG15] framework, the FFTW library, internal projects targeting embedded systems, and our new low-latency OpenVX implementation.

CHAPTER 10

Applications and Experimental Results for a Clustered Manycore Processor

In this chapter, three embedded and high-performance applications are presented. They have been implemented to evaluate the two runtimes introduced in Chapters 5 and 6. Using these two runtimes, the three applications have been implemented by hand onto the targeted clustered manycore architecture. We compare the results with the state-of-the-art using the new contributions of both applications and runtimes.

The chapter focuses on parallelization methods to exploit efficiently clustered manycore architectures like the Kalray Multi-Purpose Processor Array (MPPA)® processor. Each Compute Cluster (CC) is an omniscient multi-core Central Processing Unit (CPU) that is aware of the current state of the application running in parallel in the other CCs. This model is similar to the Bulk Synchronous Parallel (BSP) execution model [KEHS+15] (flat model). It avoids the classical master/slave approach. Indeed, the master can quickly become a bottleneck when fine-grained parallelism is required, as seen in Chapter 6.

The first application (Section 10.1) is a 3D-stencil used in numerical simulations, for instance, fluid simulation for the weather forecast, wind, and ocean. The second one (Section 10.2.1) is the Fast Fourier Transform (FFT), that is used in most signal processing applications. We show in this chapter the first implementation of a distributed Fast Fourier Transform (FFT) implementation on MPPA®. The third application, currently part of the Kalray Neural Network (KANN) framework, is a distributed runtime for executing inference Convolutional Neural Network (CNN) applications at low-latency.

10.1 Macro Pipeline for the Computation of a 3D Stencil

Section 10.1.1 presents the application. Section 10.1.2 introduces an implementation of the targeted 3D stencil application. Section 10.1.3 explains the contribution for optimizing this 3D stencil application.

The presented work is part of the Ph.D. thesis of Minh Quan Ho, who wrote the new implementation, and myself for the communication runtime, the debugging and technical discussions.

10.1.1 Lattice Boltzmann Method (LBM) Algorithm and Background

The 3D-stencil is a Lattice Boltzmann Method (LBM) used in simulations. The LBM is widely used in computational fluid dynamics for incompressible and weakly compressible flows as explained in paper [Suc01]. For instance, LBM algorithms are used to simulate oceans, failure of nuclear reactors, and volcano eruptions.

A lattice is a structured geometrical shape composed of points or objects in space.

Such stencil applications cannot fit in the local memory and must be stored in the main memory (external).

An LBM model is characterized by its stencil type, denoted DdQq, where d is the number of space dimensions (one, two or three) and q is the number of Particle Distribution Function (PDF), as explained in [CCJM97]. Physically, an LBM time step on a lattice node is broken into a *collision* and a *propagation* step, also known as the *streaming* step.

The collision applies a predefined physical model on the lattice distribution vectors. The propagation updates these new distribution values for each node. The most used stencil types are D2Q5, D2Q9, and D3Q19 (see Figure 10.1) or D3Q27. The nodes are the points from 1 to 18 and the center point in Figure 10.1. Nodes can also be found at the corners of the cube, but they are not dependencies for the D3Q19 stencil, they are for the D3Q27 stencil.

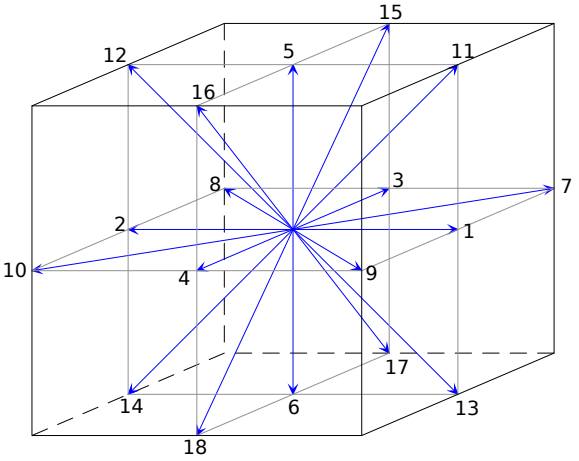


Figure 10.1 – *LBM D3Q19 Stencil*

10.1.2 Implementation State-of-the-Art

In this section, we explain an implementation of a 3D-stencil onto a Direct Memory Access (DMA)-enabled manycore processor. From a programming point of view, LBM applications are easy to implement and well-suited for parallelization on modern multi-/manycore platforms. However, LBMs are known for their low arithmetic intensity, and their high memory bandwidth. Indeed, the entire data set, usually a gigabyte of data, has to be streamed in the processor most of the time for computing a single step of the simulation algorithm. The execution time highly depends on the memory access schedule and the theoretical bandwidth of the main memory (external memory).

The initial implementation uses the Kalray OpenCL data parallel programming model presented in Section 3.2.2. The Kalray OpenCL model uses the Distributed Shared Memory (DSM) (see Section 2.4.5) to support direct memory access to the off-chip Double Data Rate (DDR) memory. However, such direct memory accesses using the DSM are synchronous,

and cache effects are also a bottleneck. The bottleneck is due to the software managed cache, aliasing conflicts, and the small number of pages that can be cached in the local memories (on-chip memories).

Hence, using explicit asynchronous transfers between off-chip and on-chip memories aims to boost performance, to reduce the overheads and the execution time. However, it implies important code re-structuration, as well as new communication primitives and algorithms.

Most existing LBM implementations on Graphics Processing Unit (GPU) employ the fused two-lattice approach as the most comfortable and most computationally efficient method. In particular, OpenCL Processor Array LBM (OPAL) from [OTK15] implements a one-step two-lattice 3D LBM solver based on the D3Q19 stencil. OPAL is designed to be portable and straightforward on GPUs, accelerators, and other OpenCL-enabled devices.

10.1.3 Optimizing a 3D LBM Stencil Application on Top of Asynchronous One-Sided (AOS)

We take the D3Q19 LBM application from OPAL presented in paper [OTK15] as a reference configuration. The data dependencies are shown in Figure 10.1. We propose in this section a generic 3D LBM streaming algorithm with domain decomposition. We detail index and halo size calculation for any configuration of the stencil distribution in Figure 10.3. For each iteration, the entire data set is streamed in the local memories. As for the OpenCL implementation, this sequence is repeated as many time as required (fixed number of iterations).

We optimize memory accesses by pre-fetching 3D tiles using Algorithm 14. A 3D tile is a convex polyhedron bounded by six quadrilateral faces. Moreover, other optimization parameters can be tuned, even on other architectures, such as the DdQq, the halo size, and the number of time steps T for the LBM simulation.

Supporting 3D Data Transfers in the AOS Library

Efficient 3D-stencil computations on a DMA-enabled processor implies 3D data movement support at low-level. Therefore, the AOS Application Programming Interface (API) of Chapter 5 has been slightly extended to support generic 2D and 3D data transfers easily.

2D transfers directly use *Put* and *Get* primitives presented in the Section 5.3.6. In the case of 3D buffers, new 3D transfer primitives based on *Put* and *Get* have been developed [JH16]. The 3D transfer primitives have been implemented over the 2D transfer ones at low-level. Indeed a 3D transfer is a sequence of 2D transfers. This is basically a "for loop" over the 2D transfer proposed by Algorithm 5 and illustrated in Figure 5.5.

Implementation Architecture

Firstly, the LBM kernel of OPAL is rewritten in C99 code. Given the similarity between OpenCL-C and standard C99, the porting process is easier. Therefore, the one-step two-lattice method is re-applied using a *pull* scheme, as in the original OPAL. The *pull* scheme means that the CPUs use remote reads (*Get*) and remote write (*Put*) to access the data.

Secondly, two instances of the 3D lattice grid (*LatticeEven*, *LatticeOdd*), each containing $L_x \times L_y \times L_z$ nodes, are allocated in the main memory (external) and are accessed in a node-wise layout. For consistency, the distribution values of a lattice are stored consecutively.

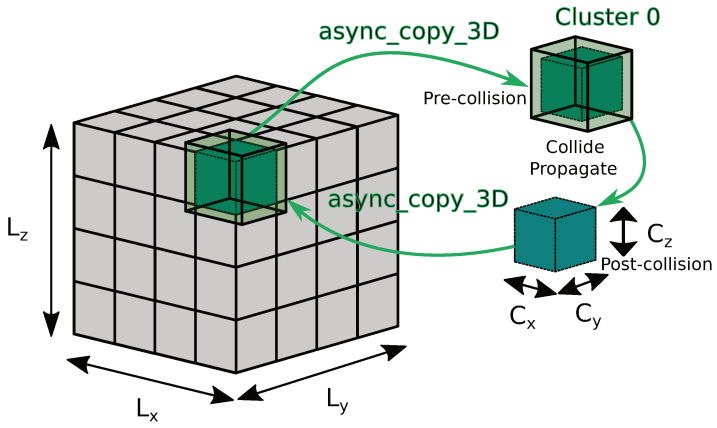


Figure 10.2 – *3D LBM/stencil decomposition where a Main-node subdomain (green) is copied with its surrounding halo layers (if exists) and one extra subdomain (blue) is needed to store the post-collision state.*

Finally, an iteration of the stencil divides the lattice domain into subdomains as seen in Figure 10.2. We copy the 3D tiles and compute the subdomains one by one in the local memory of the 16 CCs.

Each subdomain is defined as a 3D tile of $C_x \times C_y \times C_z$ nodes. To avoid repetitions, we use the subscript d as a symbol for the three Cartesian coordinates (x, y, z). Any variable or equation whose variables are subscripted by d should be interpreted as three variables or equations with x-/y-/z-subscripted terms respectively.

Dealing with the 3D Tile Pattern

We use the overlap tiling technique to deal with the data dependencies. For the sake of simplicity, we assume that L_d and C_d are powers of two and define M_d the number of subdomains in each dimension ($M_d = L_d/C_d$). The total number of subdomains is the product of the number of subdomains in each dimension $M = M_x \times M_y \times M_z$. Besides, we denote the constant $F_d = C_d + h$ to be the extended subdomain size with halo layers (h) added[1]. Thus, to update a subdomain of $C_x \times C_y \times C_z$, an extended 3D tile $F = F_x \times F_y \times F_z$ is loaded in the local memory of the CCs from the main memory (external).

This is true for most cases (non-boundary subdomains, for instance, subdomain 4 in Figure 10.3). On boundary subdomains (for example in subdomains 0, 1, 2, 3 in Figure 10.3), the extended 3D tile should be adjusted by applying a halo cutoff to deal with solid nodes. Therefore, a local subdomain slot must be allocated for $F_x \times F_y \times F_z$ nodes to match any cases.

In the implementation, the management of the boundary conditions is quite complicated because of the explicit Remote Direct Memory Access (RDMA) communications, the local memory allocation, and the indexes of computation.

Macro Pipeline of 3D Transfers

Algorithm 14 sums up the operations performed by each CC. The work is equally distributed on each CC. As such, the CCs update M subdomains within an LBM time step (a single simulation step). We define S as a fetched subdomain in the shape of a 3D tile,

[1] $h = 2$ with the D3Q19 stencil.

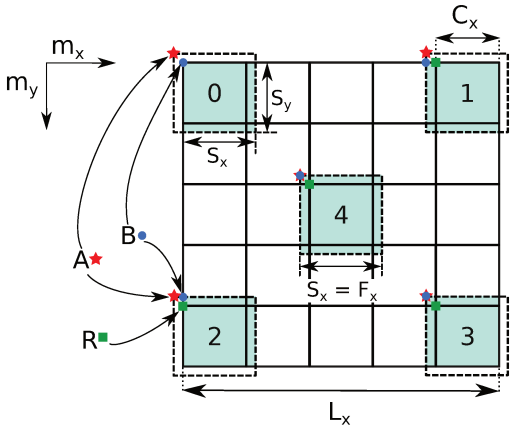


Figure 10.3 – *Local/Remote copied index in 2D (in lattice node) with A: beginning of the local buffer = (0,0); R: beginning of the remote main node 3D tile (without halo); B: beginning of the copied 3D tile (S), represented by: B_a: index of S on local memory (from A) and B_r: index of S on main memory (from R).*

and S' as the written subdomain (output) also in a shape of a 3D tile. These subdomains are organized in a *macro-pipeline* using asynchronous 3D transfer primitives which overlap computation and communication (see the Section 10.1.1).

We apply the two-lattice method to the local memory. The number of buffer slots is doubled, one for fetching the pre-collision 3D tile (S) (see Figure 10.3) from the first global lattice grid and one for storing the post-collision 3D tile (S') that will be put in the second lattice grid.

Algorithm 14 Explicit macro-pipeline of 3D-stencil updates using double-buffering within a time step.

```
1:  Input: NB_CC, M, Input_Data
2:  Ouput: Output_Data
3:  M_CC = M / NB_CC              // Number of 3D tiles
4:  K = M % NB_CC                 // Remaining 3D tiles
5:  /* Prolog: get first subdomain */
6:  prefetch_cube(0)
7:  /* Kernel pipeline: M is the number of 3D tiles per cluster */
8:  for i in 0 .. M_CC + K - 1 do
9:    if i < M_CC + K - 1 then
10:     prefetch_cube(i+1)          // get next 3D tile
11:   end if
12:   wait_cube(i)                  // wait current 3D tile
13:   compute_cube(i)               // compute current 3D tile
14:   put_cube(i)                   // put back to main memory
15: end for
16: /* Epilogue: Wait for last put and barrier the clusters */
17: RDMA_Fence()                   // Multi-clusters Read-After-Write dependencies
18: Synchronize_all_cluster()
```

The pre-collision 3D tile is allocated for $F_x \times F_y \times F_z$ nodes, while the post-collision 3D tile only needs to store $C_x \times C_y \times C_z$ nodes. Figure 10.2 only draws one global lattice grid

	Prol-ogue	m=0	1	2	3	4	5	6	7	Epil-ogue
		i=0	1	2	0	1	2	0	1	
buf[0]	G	WCP	WG		WCP	WG		WCP	W	
buf[1]	G		WCP	WG		WCP	WG		WCP	W
buf[2]		G		WCP	WG		WCP	WG		

Table 10.1 – *3-depth pipeline (triple-buffering) which allows a 2-step distance between* `GET` *and* `WAIT`*, but only a 1-step distance between* `PUT` *and* `WAIT`*, thus the* `PUT` *transfer will not be well overlapped (m: index of subdomain to compute, i: index of local buffer slot;* `G = GET; P = PUT; W = WAIT; C = COMPUTE; WCP = {WAIT + COMPUTE + PUT}; WG = {WAIT + GET}`*).*

for compactness; however, the local post-collision 3D tiles are placed in the second grid. These two global grids are then swapped before starting the processing of the next time step (for out-of-place computing in the main memory).

The algorithm uses multiple CCs at the number of NB_CC and exploits all Processing Elements (PEs) in each CC. The multi-threading is enabled using the POSIX API (`create`, `join`), introduced in Section 3.1.2). As there are 16 CCs available on MPPA®, each CC is then responsible for $\frac{M}{16}$ subdomains.

Depending on the value of M, there might be K trailing subdomains ($K \in [0..15]$). If $K > 0$, the algorithm must perform an extra step to copy, update and put back these K trailing subdomains by K CCs, while the other CCs are waiting. A synchronization barrier (Line 14) at the end of each time step is needed between all CCs to avoid *data races* at the next time step. This procedure is then repeated as many times as the number of timesteps.

The double-buffering (2-depth) pipeline in Algorithm 14 is the simplest to overlap communication and one compute-step.

As the computation is faster than the data transfers, deeper pipelines such as triple- or quadruple-buffering provide better overlapping of the communication and the computation. However, deeper pipelines require more local memory space in the CC.

In Figure 10.1, we show how a 3-depth pipeline behaves. Note that the time spent in `GET` and `PUT` is considered negligible (non-blocking) and transfers are executed in the background. However, the time spent in `COMPUTE` depends on the speed of the core, while the `WAIT` time depends on how fast the memory system serves the RDMA transfer requests and how they are hidden entirely or partially by the `COMPUTE` function.

Local and Remote Management of Copy-index

Here, we present the computation of the copy indexes, subdomain sizes, and halo cutoff managements depending on geometric positions of the subdomains. Adding a *ghost layer* surrounding the computational domain is a common technique to simplify the implementation of the streaming step at boundary cells, as seen in [MHTR08]. However, we choose not to use this approach in our work, mainly to minimize the main memory allocation and avoid wasting bandwidth and storage for moving ghost cells.

However, in our 3D decomposition algorithm, this decision requires careful calculation of copy parameters for the subdomain indexes. The pre-collision 3D tile S embeds two additional halo layers for each dimension (F_d). Its computational space begins at $(1, 1, 1)$ and ends at $(F_x - 2, F_y - 2, F_z - 2)$ included. Halo layers are shown in Figure 10.2. They are the 3D tiles placed all around the internal darker green 3D tile that is copied into the local memories of the CC. When fetching a non-boundary subdomain (main block + halo) from main memory to S, the arrival point of data at the local buffer is set to $(0, 0, 0)$, and

the remote point is computed as the global beginning position of the subdomain minus one (back-off) in each dimension ($(m_d \times C_d) - 1$).

A boundary subdomain can have up to three missing sides, depending on its location, as it can be observed in Figure 10.3. Consequently, the halo layer of these missing sides needs to be pruned from the copied 3D tile. The remote read-point and local write-point must also be adjusted as well.

10.1.4 Results and Discussions

Benchmarking Environment

The pipelined 3D LBM algorithm is implemented on the MPPA® second generation platform using POSIX threads (see Chapter 6) and asynchronous 3D primitives from AOS (see Chapter 5). By default, MPPA®-256 cores are set to run at 400 MHz, and LP-DDR3 frequency is configured at 1066 MHz, i.e., ∼8.5 GB/s peak per DDR.

Note that MPPA® embeds two DDR interfaces (North and South) and the current OpenCL runtime only uses one DDR and exposes 1 GB of available main device memory, while the MPPA® AOS library exposes both single and double-DDRs modes.

Different cubic cavity sizes, varying from 64 to 224, are used in our tests, with some exceptions. Problem sizes larger than 160 cannot be run in OpenCL on MPPA® due to the 1 GB device memory limit (OpenCL provider's limitation). Local work-group size in OPAL OpenCL is always set to $32 \times 1 \times 1$, as it delivers the best performance in most of the cases.

In single-DDR mode (POSIX and OpenCL), both *LatticeEven* and *LatticeOdd* are allocated on the North DDR. In double-DDRs mode (POSIX-only), the *LatticeEven* buffer is allocated on the North DDR, and the *LatticeOdd* is on the South DDR.

The effective throughput of the double-DDRs mode can be considered as twice as one of the single-DDR mode; thus 2× performance is expected.

We present below execution times of the OPAL kernel rewritten with our new POSIX pipelined algorithm on the MPPA®-256, called *OPAL_async*, in 3-depth and 4-depth pipelines, and following the local two-lattice method on various cavity sizes. These tests are further run in both single- and double-DDRs modes. All these runs are checked for correctness against the original OPAL code on GPU.

LBM performances are measured in Mega Lattice Updates per Second (MLUPS). Therefore, it measures the speed of the system to compute lattice objects per seconds, precisely the speed to update a point presented in Figure 10.2.

Pipelined 3D LBM Stencil on MPPA®

As seen in Figure 10.4, the OPAL_async algorithm outperforms the OpenCL version by more than 30% in the single-DDR mode (from 12 MLUPS to 16±1 MLUPS). Furthermore, we notice that the configuration with less Halo Bandwidth (HBW) (3-depth, 36% HBW) delivers higher performance than the 4-depth configuration (43% HBW). While consuming memory bandwidth, halo cells are copied because of the read-dependency between neighbors. It does not contribute to the final performance. Figure 10.4 shows that the less memory bandwidth the halo cells take up, the more performance we obtain.

Such a result leads us to think that the HBW of 2D/3D-stencil computations aimed to reach Exascale, like weather forecast, ocean simulation and computational fluid dynamics, should be lessened on future clustered manycore processors. For this to happen, these manycore chips should embed larger local memory on each CC (compute unit) to tear

down the useless part of halo exchange due to domain decomposition. Finally, Figure 10.4 also shows the expected 2× speedup for using two DDRs instead of one.

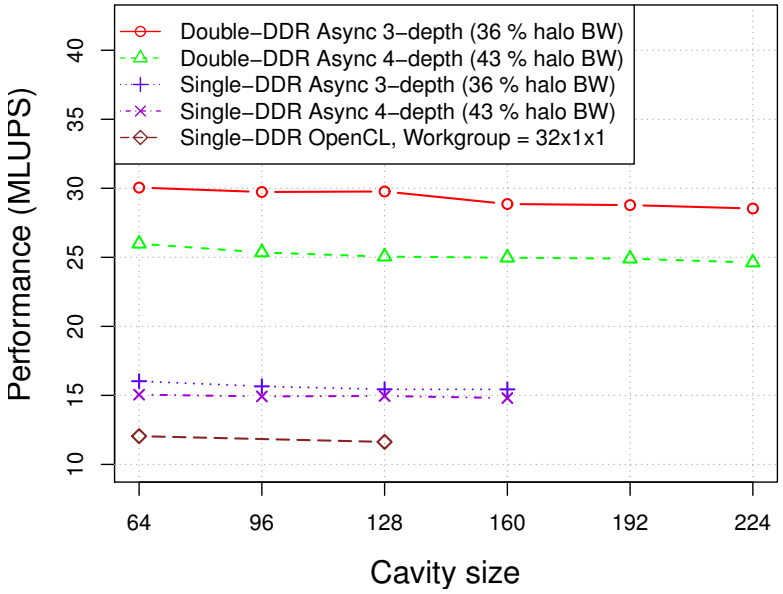


Figure 10.4 – *OPAL_async vs. OPAL OpenCL on MPPA® for duration = 1000 steps.*

Performance Extrapolation

For a better understanding of the benefit of our pipeline algorithm, we modified the OPAL_async code to be able to work with arbitrary depths. Different pipeline depths were then tried out (1, 2, 4, 6, 8), to see if increasing the number of asynchronous buffers improves the performance.

Thus, the node size is reduced to $8\times8\times8$ to make the storage of up to eight subdomains possible in the local memory of one CC. Moreover, instead of using all the 16 CCs, we now vary this number of CCs and set the domain size to 128^3 to study the strong scalability of the algorithm. We use only the double-DDRs mode to obtain the best performance.

In Figure 10.5, as expected, the 1-depth code (blue line) is slower than other versions with communication-computation overlapping. However, we obtain exactly the same performance as the double-buffering case when using more than two buffers (4, 6, 8). The performance line scales from 1 CC to 8 CCs, then reaches almost a stable value of between 20-22 MLUPS from 8 CCs to 16 CCs.

To explain this, we added the sustained throughput of 3D transfer (red line) from the Kalray unit test dedicated to 3D asynchronous copy. This test only does some ping-pong copies to the DDR and does not perform any calculation (Arithmetic Intensity (AI) = 0 flops/byte explained in the Roofline model [WWP09]). We observe that the native 3D copy reaches the maximum throughput with as few as four CCs (6GB/s), then remains the same for higher numbers of CCs. Therefore, four CCs are enough to saturate the DDR bandwidth. Unlike the 3D unit test, our LBM code performs real computation on the copied data. Its Arithmetic Intensity (AI) is about $350/(2*19*4) = 2.3$ flops/byte, which means that each CC spends more time working on a 3D data node. The result explains in

Figure 10.5 the MLUPS performance which reaches its upper bound for 8 CCs, instead of 4 CCs.

Another precise way to interpret the performance of 20-22 MLUPS is to apply the performance estimation formula presented by McIntosh-Smith & *al.* [MSBCP14]:

$$P = \frac{B \times 10^9}{19 \times 2 \times 4 \times 10^6} \ (MLUPS) \tag{10.1}$$

in which B is the effective memory bandwidth in GB/s. To take into account the additional cost of halo copy in our decomposition algorithm, we multiply P by $(1 - HBW)$, the effective part of bandwidth (main node) which generates the real performance:

$$P_h = \frac{6.0 \times 10^9}{19 \times 2 \times 4 \times 10^6} \times \frac{8^3}{10^3} = 20.2 \ MLUPS \tag{10.2}$$

The formula for P_h shows a little performance gain to perform asynchronous transfers on clustered manycore processors as for today.

However, it can be seen that the overlapping gain time is small compared to the wait time for data due to the DDR3 bottleneck. It also demonstrates the memory-bound property of general stencil computations. We think that newer memory technologies like DDR4 and High Bandwidth Memory (HBM) will be a performance boost on these architectures.

The scale-down of the 3D throughput versus the peak 17GB/s of two DDRs is caused by the fact that strided copies (2D/3D) must read data from a lot of different DDR memory banks. Furthermore, these copies are poorly aligned due to the access pattern of the application ($Q = 19$ floats). This efficiency factor of 3D transfers can be compared to the linear copies (contiguous).

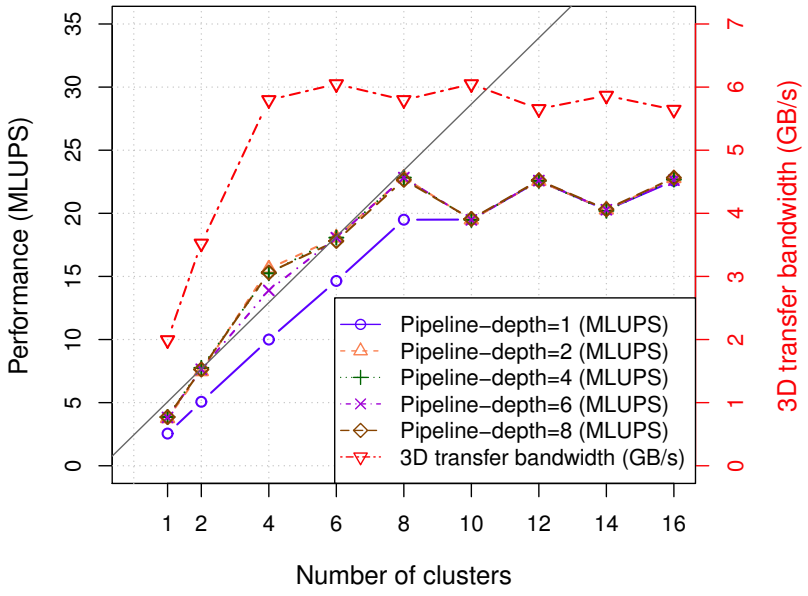


Figure 10.5 – *Performance extrapolation of OPAL_async with* $8 \times 8 \times 8$ *subdomains with the first eight CCs correlation represented by a gray line for 1000 timesteps and cavity size 128.*

A correlation, computed by the `lm` function in `R`, from 1 to 8 CCs would give the performance expectation of our streaming algorithm if we were not bounded by the memory

bandwidth (affine gray line). These results confirm that our pipelined LBM algorithm is strongly scalable, but is quickly memory-bound on MPPA®.

Indeed, the performance of the designed algorithm is limited by the hardware memory bandwidth when 8 CCs or more are used.

Our results also show that the imbalance between computing power and data throughput is one of the most substantial drawbacks of actual clustered manycore processors, and demonstrate the interest of future memory technologies with high-bandwidth.

10.2 A Low-Latency Distributed Fast Fourier Transform

In this contribution, we use multiple CCs to parallelize the well-known FFT algorithm, contrary to [HNEdD15] where the computation targets only one CC. Distributing the FFT computation over several CCs reduces the execution time; thus, its execution latency.

10.2.1 Fast Fourier Transform

Fourier analysis converts time (or space) to frequency (or wavenumber) and vice versa. Fourier analysis has many scientific applications in physics, signal processing, imaging, probability theory, statistics, cryptography, numerical analysis, acoustics, geometry, and other areas.

The Fast Fourier transform (FFT) [HJB84] algorithms compute the Discrete Fourier Transform (DFT) while reducing the complexity from N^2 to $Nlog_2(N)$. Let us consider a complex 1D array of N values. The raw DFT is defined by the following formula:

$$X(f) = \sum_{k=0}^{N-1} x_k e^{-2i\pi kf/N} = \sum_{k=0}^{N-1} x_k W_N^{kf} \tag{10.3}$$

$$W_N = e^{-2i\pi/N} \tag{10.4}$$

10.2.2 Computing Techniques of Fast Fourier Transform

The FFT algorithms [CT65] re-factor formula 10.3, and they are known as the: Radix-2, Radix-4, and the Six-Steps FFT. FFT algorithms compute the same values as the DFT except for possible rounding errors. These FFT algorithms can be used independently or combined, providing several trade-offs concerning computational complexity, memory requirement, and parallelism. The challenge here is to find the optimal combination of FFT algorithms for execution on the MPPA®. Below, we discuss the FFT algorithms which have been used or tested in this work.

Radix-2

The Radix-2 algorithm is applied to inputs whose sizes are powers of 2. Its complexity is $\frac{10}{2}Nlog_2(N)$. The Radix-2 Decimation-In-Time equation is listed below [Che00]:

$$X(f) = \sum_{k=0}^{\frac{N}{2}-1} x_{2k} W_N^{2kf} + \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1} W_N^{(2k+1)f} \tag{10.5}$$

Radix-4

The Radix-4 algorithm is applied to inputs whose sizes are powers of 4. Its complexity is $\frac{34}{8}Nlog_4(N)$. Note that by default a complex multiplication requires four multiplications and two additions. For this reason, the Radix-4 algorithm might be more suitable regarding performance as it requires fewer multiplications than the Radix-2 algorithm. The Radix-4 Decimation-In-Frequency equation is given by the following formula [Che00]:

$$X(f) = \sum_{k=0}^{\frac{N}{4}-1} \Big[x_{(k)} + x_{(k+\frac{N}{4})}(-i)^f + x_{(k+\frac{2N}{4})}(-1)^f + x_{(k+\frac{3N}{4})}(i)^f\Big] W_N^{kf} \tag{10.6}$$

Six-Steps

The Six-Steps method [SG12] is another way of computing FFTs. Whereas the Radix-2 and Radix-4 algorithms are sequential, this method provides an efficient way to parallelize the FFT computations by splitting them into smaller ones. The six steps are:

1. **Transpose**, Transposition of the matrix interpretation of the complex 1D input.
2. **Fast Fourier Transform**, Independent FFT computations provide the maximum degree of parallelism.
3. **Transpose**, Transposition of the matrix interpretation.
4. **Twiddle Correction**, Complex multiplication by each corresponding Twiddle factor on the entire complex matrix with the coefficient $e^{-2i\pi * \frac{row * line}{matrixSize}}$.
5. **Fast Fourier Transform**, Independent FFT computations provide the maximum degree of parallelism.
6. **Transpose**, Transposition of the matrix interpretation.

This algorithm provides both embarrassing parallelism and data locality during the FFT steps (2) and (5), which means that it is very suitable for parallel implementations and efficient on-chip memory usage. Moreover, this method can make the use of either the Radix-2 or the Radix-4 algorithms possible during FFT steps.

Real to Complex FFT

A real N-point FFT computation can be folded into a complex $\frac{N}{2}$-point FFT. The idea is to store at the input of the FFT computation the even part in the real indexes and the odd part in the imaginary indexes. Then the FFT is performed, and the output samples are combined together in order to extract the N-point FFT final result with the following formulas:

$$X(f) = \frac{1}{2}\Big[(x_{(f)} + \overline{x_{(\frac{N}{2}-f)}}) - i(x_{(f)} - \overline{x_{(\frac{N}{2}-f)}})e^{-2i\pi\frac{f}{N}}\Big] \quad f \in [0, \tfrac{N}{2}[\tag{10.7}$$

$$X(f) = \frac{1}{2}\Big[(x_{(0)} + \overline{x_{(0)}}) - i(x_{(0)} - \overline{x_{(0)}})\Big] \quad f = \tfrac{N}{2} \tag{10.8}$$

This process is very efficient, as it reduces the number of operations for a real N-point FFT of a real signal almost by half.

10.2.3 Distributed Fast Fourier Transform Implementation

Strategy Overview and Positioning

We position our work versus previous contributions [HNEdD15] and we explain the low-level implementation of the low-latency distributed FFT over the RDMA Network on Chip (NoC) of the Kalray manycore processor.

Unlike [HNEdD15], the new implementation distributes the work on all available CCs. In [HNEdD15], the algorithm is designed using fixed-point operations to reduce the memory footprint to fit the 2 megabytes of local memory. Such optimization is possible because the targeted project did not require floating-point precision. Moreover, despite the careful management of the fixed-point implementation, a Symmetric Multi-Processor system (SMP) implementation in a single CC is a lot simpler than a distributed implementation over several CCs. However, this step is taken in our new implementation.

Our distributed FFT implementation is also based on the six-step method. The six-step method let us distribute the work on several CCs.

To the best of our knowledge, this FFT implementation is the first to break the limitation of the single CC implementation using the SMP model. Thanks to this contribution, it is possible to run complex floating-point FFTs of size greater than 2^{16}, for instance, 2^{18} and 2^{20} (the million points FFT), on an MPPA® second generation.

Slices of the matrix of the six-step method are mapped in each CC. A constraint is that the matrix must be squared; thus, it leads to supporting FFT sizes matching the power of 4. The parallelization in the CC is managed using the same mechanism already proposed in [HNEdD15], but single precision floating-point numbers are used instead of integer in fixed-point.

Architecture of the Implementation

All steps are illustrated in Figure 10.6, which shows precisely the distribution of a 256-point complex FFT on 4 CCs. A first step consists in interpreting the input, a one-dimensional array of complexes, as a matrix. The 256-point input is interpreted as a $16 * 16$ square matrix. We slice the matrix into identical block sizes, that are copied from the main memory (external DDR memory) to the local memories of the CCs.

Secondly, we transpose the matrix using inter-cluster asynchronous RDMA transfers (stride-to-stride). Transposing a matrix distributed on several CCs requires that each CC communicates with all other CCs participating in the computation of the FFT. As such an all-to-all communication pattern has to be performed.

Then each CC executes in parallel using the SMP model independent FFTs of size $\sqrt{256} = 16$ considering our example in Figure 10.6.

We perform a transposition at CCs level. The twiddle correction is then computed. The twiddle correction multiplies each value of the distributed matrix by a corresponding complex factor computed by $e^{-2i\pi * \frac{row * line}{matrixSize}}$. The *row* and *line* are respectively the numbers of rows and lines of the interpreted global matrix. However, we computed off-line these coefficients as the transcendent functions, namely the *cos* and *sin* functions of the math library, have a significant cost.

Finally, we repeat a pass of embarrassingly parallel FFTs and a distributed transposition. The final distributed transposition keeps the sparse memory accesses on-chip, instead

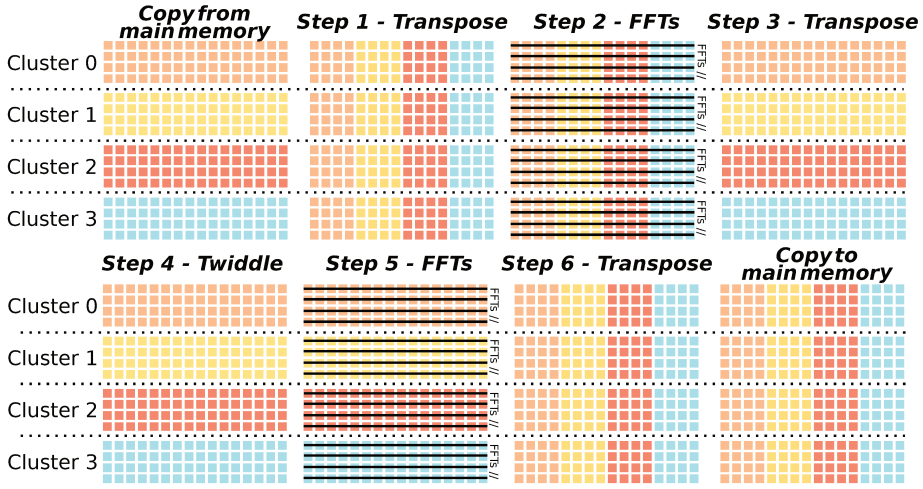


Figure 10.6 – *Architecture of the Distributed FFT for Low-Latency Execution over Several Compute Clusters (CCs).*

of the external DDR memory. Indeed, the DDR memory system has poor performance on sparse memory accesses.

On-chip Distributed Transposition

Three distributed transpose steps can be seen in Figure 10.6. The software is in charge of inter-cluster synchronizations, the pattern of the data transfers and the interleaving of the distributed communications to avoid serialization points in the NoC.

Each local memory of a CC allocates two buffers to perform the transposition. One buffer is read, and the other one is written. We either do local linear reads, and remote strided writes or do local strided reads and remote linear writes to execute the transposition operation. As such, we intensively use on-the-fly data restructuring supported by AOS (see Section 5.2.2).

The proposed distributed transposition is shown in Algorithm 15. For simplicity, the algorithm only performs local reads and remote writes (Async. Put at Line 13). Transfers are interleaved, as each CC starts by transposing locally the diagonal of the distributed matrix (only *Load/Store operation*) using optimized streaming memory accesses, and then the CC sends sparse data to their identifier $(cid + 1)\%NB\_CC$.

A posted remote atomic operation is sent at the end of the communication from one CC to another. Everything is done asynchronously for performance, but the ordering of RDMA operations is guaranteed at the end when waiting for the completion of the global transposition (at Line 17). The AOS library (see Chapter 5) guarantees the ordering.

Local Transpose Optimization

When using either 1 CC or multiple CCs for transposing the diagonal blocks of the distributed matrix, a *Load/Store*-based transposition is used. The feature of matrix transposition is that it implements sparse memory accesses that behave poorly regarding the caches. In the CC of MPPA®, the PEs contain a blocking L1 data cache. It implies that moving sparse data without any prefetching mechanism is very inefficient. Instead, we use streaming memory accesses explained in Section 4.5.2.

We manually unroll the internal transposition loop, we use explicit streaming memory accesses using streaming double-word (1-complex-float (single-precision) = 2-float-number

Algorithm 15 Out-of-place Distributed Transposition Algorithm Operation on All CCs

```
1: Input: In_M, T_W, T_H, NB_CC
2: Output: O_M
3: Synchronize all compute cluster  // Weak synchronization, no ordering
4: cid = ID current cluster         // ID in range 0 and (NB_CC-1)
   // Interleave RDMA transfer across clusters
5: for j in cid ... (NB_CC - 1 + cid) do
6:    if j == cid then
7:       Optimized local transpose      // Streaming memory accesses
8:       Continue                       // Go next iteration
9:    end if
10:   for i in 0 ... (T_H- 1) do
11:      Size = size of a float-complex
12:      target_cid = cid % NB_CC
13:      Async. Put (
         In_M+Size*T_W/NB_CC*j+Size*j, // Local address
         j,                            // Target cluster ID
         O_M+Size*T_W/NB_CC*cid+Size*T_W*j, // Remote address
         Size,                         // Object size
         T_W/NB_CC,                    // Number of objects
         Size*T_W,                     // Local stride
         Size)                         // Remote stride
14:   end for
15:   Async. Postadd to CC of ID j    // Asynchronous posted remote atomic
16: end for
17: Wait NB_CC - 1 contribution of remote atomics
```

= 8 bytes) loads and stores. The internal loop issues 16 sequential streaming loads, and then 16 sequential streaming stores, in the same order as they were loaded. In theory, this makes it possible to move 8 bytes per cycle without any PE stalls on the memory hierarchy. In practice, if there is no memory contention, aligned memory accesses, and no Read-After-Write (RAW) core stalls due to poor software implementation, moving 8 bytes per cycle is possible.

The implementation provides a speedup of almost 2 compare to the cached *Load/Store* one, on a single-core. A matrix of size $256 * 256$, at 500 MHz, is transposed in 0.57 millisecond (3.7 bytes/cycle). The streaming-based implementation is 0.31 millisecond (6.7 bytes/cycle). In the CC, the streaming *Load/Store* pipeline has an efficiency of 84 % of the peak hardware memory throughput. Results are independent of the matrix content, and the read and write buffers are invalidated by software at the L1 cache level before transposing the matrix. The measurements are performed using an average of the execution times over a thousand executions.

Memory Footprint

Each CC, part of the parallel processing of the distributed FFT, allocates two buffers for the FFT data elements. These two buffers are used to deal with the matrix transposition which relies on double buffering. Otherwise, fined-grained multi-cluster synchronizations are required; that is very time consuming for MPPA®. Indeed, each synchronization costs at least 2.2 microseconds (see Chapter 5).

Efficient fined-grained synchronizations would make it possible to remove the second transposition buffer and instead use far smaller buffers to accommodate temporary tiles for the transposition. Many synchronizations are needed to avoid data races.

Moreover, the twiddles of the twiddle correction of the distributed six-step FFT are precomputed. However, we only compute the first twiddle of each row and the rotating twiddle, to make on-line twiddle computation possible using the homothety math technique. This optimization reduces the memory footprint.

Then if we consider a distributed FFT in blocks of size $Size$, the small FFTs run in each PE of the CC is of size $\sqrt{Size}$. Thus, we also compute off-line the twiddles of this small FFT. These twiddles are quite light in terms of memory footprint. For instance, considering a million point FFT, the size of the FFT run in each PE is only 1024-point.

A summary of the formula to be used to compute the memory footprint in each CC is given in Table 10.2. Thus, using 16 CCs, the biggest FFT size with local memory (on-chip memory) of 2 megabytes is $2^{20} = 1048576$ as the memory is $8*2*1048576/16+\sqrt{1048576}*8*2*2/16+\log_2(\sqrt{1048576})*8*\sqrt{1048576} = 1132544$ bytes.

Buffer Types	Distributed FFT of Size $Size$ (bytes)
Transpose Two Buffers	$8*2*Size/NB\_CC$
Distributed FFT Twiddles	$8*2*2*\sqrt{Size}/NB\_CC$
Local FFT Twiddle	$8*log_2(\sqrt{Size})*\sqrt{Size}$

Table 10.2 – *Summary of the Memory Footprint of the Distributed FFT on Several CCs*

Floating-point Single Instruction, Multiple Data (SIMD) Instructions

The k1 Very Long Instruction Word (VLIW) core implements SIMD operators to process efficiently single precision complex multiplications. Indeed a complex multiplication is defined as: $(A_r + iA_i) * (B_r + iB_i) = (A_rB_r - A_iB_i) + i(A_rB_i + A_iB_r)$. The real part, $C_r = (A_rB_r - A_iB_i)$, is computed using a floating-point instruction implementing a dual multiply and subtract in one cycle. The imaginary part $C_i = (A_rB_i + A_iB_r)$ is obtained using a floating operator with cross dual multiply and addition. The register width of the k1 VLIW core is 32-bit. A single precision floating-point number can fit. Moreover, for 64-bit SIMD operations, it is possible to work on register pairs (Section 2.3.2).

Figure 10.7 shows the elementary vectorized instructions to perform a complex multiplication on the k1 VLIW core. As the compiler usually has difficulties to generate such instructions, we wrote the code using inline assembler, as it avoids dealing with the register allocation, the stack management, and the Application Binary Interface (ABI). The ABI is a hardware-dependent format that defines the rules to manage data structure, calls to routines in a machine code point-of-view. Using inline assembler or intrinsics still, let the compiler optimize what is placed. Intrinsics would have been better, but they are not supported by the GNU Compiler Collection (GCC) toolchain in this case. The $fcma$ (Floating-Point Cross Multiply-Add) and the $fdms$ (Floating-Point Multiply-Subtract) provide 3 floating point operations each per cycle. Such SIMD optimizations make a theoretical speedup of 3 possible, but in practice, it is not the case because of memory hierarchy stalls and the *Load/Store* Unit bandwidth.

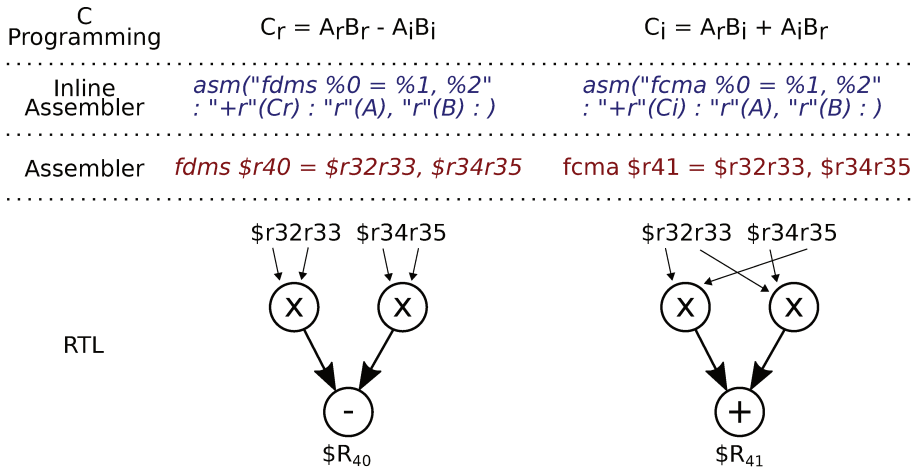


Figure 10.7 – *Example of a Vectorization (pair of registers) in the k1 VLIW PE.*

10.2.4 Results & Discussions

Figures 10.8, and 10.9 shows experimental using both the one CC and the multi-clusters implementations. Small FFTs are not run on the multi-clusters implementation as the initiation of the NoC communications last longer than the computation. Therefore, the performance decreases when we increase the number of CCs. For instance, it can be observed in the multi-clusters implementation (see Figure 10.9) targeting the 65536-point use-case using 2 CCs. We observe a performance drop because of the inter-cluster RDMA operations. As such the mono-cluster benchmark shows strong scaling (see explanation in Section 7.4.1) when varying the number of PEs. The multi-clusters implementation shows strong scaling when varying the number of CCs with a fixed number of PEs inside each CC. In the multi-clusters benchmarks, the number of PEs is set to 16.

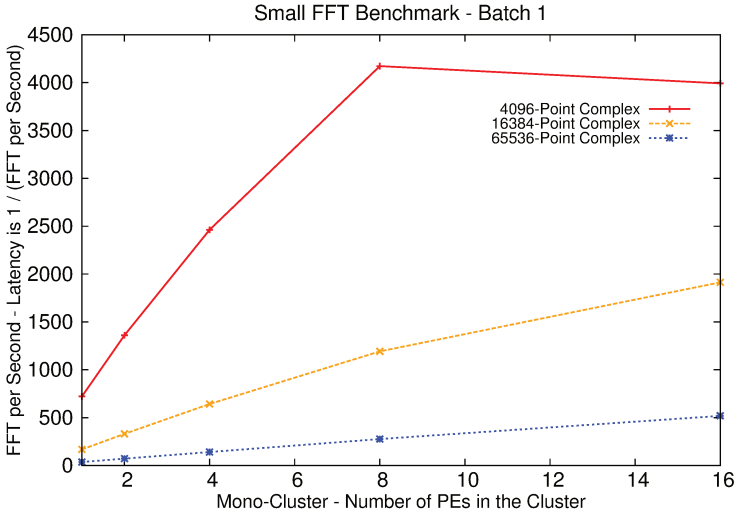


Figure 10.8 – *Execution Time of the Mono-Cluster FFT. The Higher, The Better.*

Also, the mono-cluster implementation (see Figure 10.8) provides quasi-linear speedups for FFTs of size 2^{14} and 2^{16}. The speedup of the FFT with size 2^{12} clamps at 8 PEs. Fair speedups are observed for the multi-clusters implementation, but the main issue is the transposition over the NoC using RDMA operation with the AOS API. For instance, the

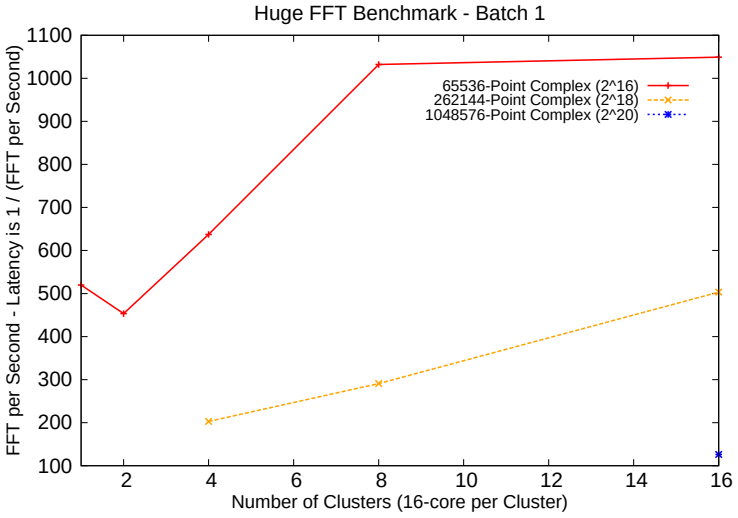


Figure 10.9 – *Execution Time of Distributed Multi-Cluster FFT. The Higher, The Better.*

million-point FFT the transposition steps take more than 65 % of the total execution time (126 FFT/second of 2^{20}-point).

Limitations & Conclusion

With this new implementation, the largest FFT is a million single-precision complex points (2^{20}) on a full MPPA® processor, This is currently a limitation, and some work is already on-going to use the main memory (external). The applied technique is the same as the one used in the OpenVX framework (see Chapter 9).

Using the main memory (external) is possible thanks to the contribution of Chapter 5 that drastically decreased the complexity of programming the MPPA® processor along with providing performance. However, AOS has its limitation that is well seen in such benchmarks. Indeed, in this FFT implementation, fined-grained communications over NoC are required, making the implementation behave poorly for small FFTs.

However, this implementation is the first to break the limit of 2^{16} floating-point FFT limit for executing on the MPPA® processor. We show that an all-to-all communication pattern, implemented by the flat distributed matrix transpose, is possible. Finally, FFT, and in particular huge FFTs, are known to be complicated to implement efficiently due to the irregular memory access patterns. They are even more challenging on distributed local memory architectures like MPPA®, where all data movements, computation distributions, and synchronizations are performed by software (DMA).

A version of this FFT is proposed and available on GitHub. Usage requirements are provided on the GitHub website: *https://github.com/kalray/Benchmark_FFT*

10.3 Distributed Runtime for CNN Inference

CNN applications are state-of-the-art for today computer vision and artificial intelligence applications. In the future, CNNs will be unavoidable for embedded computing. CNN applications are both data and compute intensive. Thus, CNNs are the typical benchmarks to evaluate embedded computing platforms for car manufacturers and high-performance systems.

This work is part of a proof-of-concept for executing CNN applications in inference mode on the MPPA® processor. In this use-case, we target low-latency (fast) execution of CNN applications on an embedded Multiprocessor System-on-Chip (MPSoC). This proof-of-concept is today known as the KANN framework [Kal]. KANN is the Kalray's neural network framework that makes it possible to run efficiently deep learning networks on MPPA®. It supports many classical CNNs like GoogLeNet, AlexNet, SqueezeNet, and others.

The KANN framework is divided into three parts. The first part is our contribution: the distributed runtime for low-latency execution of CNN applications in inference. The runtime is written in bare-hypervised level to reach maximum performance as explained in Section 2.4.5. A second part, by Frederic Blanc, is the code generator that takes as an input the CNN structure. The last part, by Julien Lemaire, is the optimization in assembler or inline assembler of the different pools of kernels required by classical CNNs. For instance, the convolution kernels are usually responsible for 80% of the execution time, the max-pool, the fully-connected and the local response normalization kernels.

10.3.1 General Architecture

We explain our architecture of the runtime and the static software synthesis tool. The architecture is shown in Figure 10.10, where the steps from the CNN description to the execution of the MPPA® processor are described.

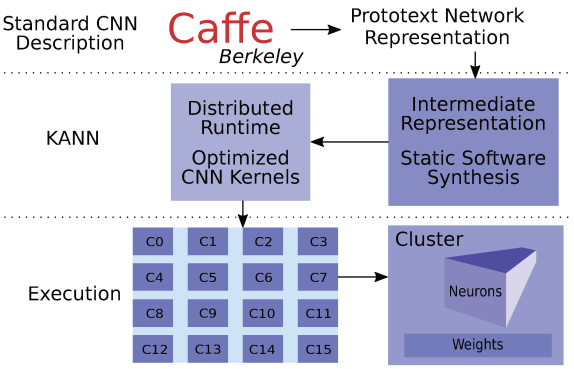


Figure 10.10 – *Architecture of the KANN Framework.*

We take as input a standard *Caffe* [2] CNN description and we convert it into a proprietary intermediate representation to apply transformations like neuron and weight distributions. Neurons are the input data of the CNN application and weights are static constants that are read at each iteration of the CNN network. They are used in the layers representing the neural network.

10.3.2 CNN Runtime for a Clustered Manycore

In this section, we present the distributed runtime for executing CNNs targeting low-latency on the MPPA® processor and the multicast engine.

Distributed Runtime

Inspired by the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) tool presented in Chapter 7, our distributed CNN runtime eases the deployment handling

[2] http://caffe.berkeleyvision.org/

of automatic RDMA communications and multicast operations. The code generator provides information like the number of CC used, and the local memory size to be allocated in each CC to execute the CNN inference.

Here, we use the MPPA® in acceleration mode over the Peripheral Component Interconnect Express (PCIE), with an external Intel®host x86. Each CC implements a function handler that is executed for running the generated code. We list, in execution order, the performed steps to deploy and enable such a back-end runtime for the MPPA® processor.

- The x86 host boots the MPPA® over the PCIE, and the controller of MPPA® creates control queues to communicate with the host.
- The controller allocates the array of the weights and input for neurons (usually an image) in the main memory (external), and notifies the host over PCIE queues. Then, the host x86 sends the weights of the CNN application.
- The controller initializes the distributed runtimes like the AOS library (see Chapter 5) and our multicast engine to send the weights efficiently to the CC.
- Each CC initializes its runtime, allocates an array of memory to store copies of the input neurons and the weights (see Figure 10.10).
- Each CC and the controller creates and clones the memory windows (see Chapter 5) to make each node able to access the memory of other nodes. The memory window is defined as AOS segments where RDMA *Put* and *Get* operations are performed. The communication paths are used for inter-cluster data transfers to satisfy the data transfers/exchanges of the mapped sequence of CNN layers.
- When the system is ready and synchronized, the input data are sent to the main memory (external), the generated code is executed, and finally, the output data is copied to the x86 host.

Multicast Engine

CNN implementations copy all weights (network parameters) in the on-chip memory of the System-on-Chip (SoC) in charge of computing the CNN inference.

The Kalray NoC supports multicast operations, handled by software. We wrote a DMA micro-code that reads linear buffers (same as Algorithm 5 but for linear transfers), and we computed custom multicast routes to deliver the data elements to the proper CCs. Multicasted elements are delivered on the CC side on a preconfigured and ready Rx data tag with an end-of-transfer (End-of-Transfer (Eot)) for the completion on the DMA NoC interface of this CC. Figure 10.11 shows the topology of the multicast data transfers onto the CCs. Such a data communication scheme reduces the transfer time by 3. Thanks to this contribution, we measure a cumulated in-coming data bandwidth of 24 gigabytes per second in the 16 CCs using a single DDR3.

As seen in Chapter 5, using 16 CCs the best unicast measured bandwidth is 8 gigabytes per second for the main memory (external DDR3).

10.3.3 Results & Comparisons

We compare here our MPPA® CNN framework implementation to the Nvidia Tegra X1 Maxwell architecture, using the famous GoogleNet [SLJ+15] network in inference mode.

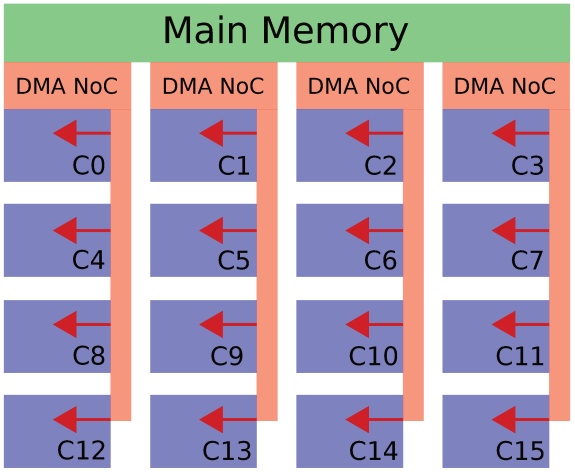


Figure 10.11 – *Broadcast Operation From the Main DDR3 Memory to the CCs.*

The two embedded MPSoC are the MPPA® second generation that was taped-out in 2015 with TSMC 28*nm* technology and the Nvidia Tegra X1 in 20*nm* technology, also released in 2015. The Tegra X1 architecture implements 4 ARM Cortex-A57 cores as a host CPU, and 256 CUDA cores, using a DDR4 controller for main memory accesses. The MPPA® second generation uses DDR3 technology. Benchmarks for the Nvidia Tegra X1 have been run and verified, attesting results of [CPC16].

Therefore, it can be observed in Table 10.3 that our new MPPA® implementation provides competitive throughput with respect to the Nvidia embedded Maxwell GPU. In the benchmark we measure execution times. Therefore, we use only the batch-1 use-case (no pipeline for input neurons). Batch-1 implementations put more pressure on the main memory as all parameters have to be copied in the SoCs for every single iteration of the CNN. In any case, the MPPA® low-latency inference implementation outperforms the embedded MPSoC of Nvidia in speed and energy efficiency.

Architecture & Configuration	FPS	Watts	FPS/Watt
MPPA® 400 MHz	47	15	3.1
MPPA® 500 MHz	56	17	3.3
Nvidia Tegra X1 Maxwell 690 MHz	33	11	2.9

Table 10.3 – *Performance of the GoogleNet CNN batch-1 (latency = throughput) using Single-precision Floating-point Operation*

10.4 Conclusion

We use here three applications, namely the 3D-stencil based on LBM numerical simulation, the FFT and a distributed runtime for the execution of low-latency CNN inference.

For the 3D-stencil application, quasi-linear speedups are obtained until 8 CCs (strong scaling benchmarks). The 3D-stencil becomes bounded by the main memory bandwidth with more than 8 CCs. Indeed stencil applications like the LBM are known to feature low Arithmetic Intensity (AI) making the application likely to be bound by the main memory bandwidth. Although the application is memory bound, our new MPPA® implementation of the LBM still outperforms the original OpenCL implementation by 33 %. The computation and communication steps are overlapped, and it is highly energy-efficient.

The FFT implementation shows linear speedup inside one CC and near linear speedup when distributing the computation of several CCs. FFTs are known to have non-contiguous memory access patterns. The six-step method splits an FFT into many smaller FFTs and provides embarrassingly parallelism and data locality during FFT steps. However, this method needs at some points a global transposition of the data set. When using multiple CCs, the overhead of the software implementation of inter-clusters RDMA transfers is significant, and it drastically bounds the speedup.

As this bottleneck has been identified, future generations of the MPPA® processor will accelerate DMA NoC configurations. Such a feature will reduce the inter-clusters communication time; and thus, speed up the overall timing latency of the application.

The distributed runtime for the execution of low-latency CNN applications is part of a collaborative work done with two Kalray's engineers. The targeted use-case is the GoogLeNet CNN, in inference mode. We show that the MPPA® processor features enough computing power considering both memory bandwidth (off-chip, on-chip, multicast) and processing with SIMD and streaming memory accesses, to compete with Nvidia's Maxwell embedded GPU.

Despite the hardware technology differences, for instance, the $20nm$, the DDR4 and 690 MHz operating frequency for the Nvidia GPU and the $28nm$, the DDR3 and the 500 MHz for the Kalray MPPA®, the final implementation is competitive thanks to the VLIW cores that provide efficient Instruction-Level Parallelism (ILP), the inter-cluster asynchronous RDMA transfers, and the multicast operations (avoid the bandwidth wall).

CHAPTER 11

Conclusions

The programming of clustered manycore processors with distributed local memories is challenging, especially when these local memories are small. However, such machines are serious competitors concerning low-power computing. Explicit software communication is painful, engineering time consuming, fastidious, and error-prone. That is why automatic tools and software runtime that carry out all of these issues are required.

In this thesis, we proposed novel techniques applied to clustered manycore architectures like the Kalray Multi-Purpose Processor Array (MPPA)® that is at the center of the work. The presented techniques leverage the challenges of data movements and explicit parallelism at the metal level on the MPPA® processor. As the MPPA® processor is a local memory based architecture implementing a partitioned global address, it makes our contributions even more challenging to be designed, implemented, debugged and validated. We also proposed higher level tools and frameworks to make the programming of the MPPA® processor more accessible using both static and dynamic dataflow models. Dataflow models provide the user with an intuitive way of expressing application parallelisms, that can enable many optimizations, thanks to the model analyzability. Finally, we presented a distributed framework for the execution of OpenVX graphs at low-latency, and also, low-level implemented parallel applications running on the Kalray MPPA® processor.

Throughout this work, one of the most difficult thing to perform when developing in a massively parallel environment is understanding why it fails. As such, the complexity of the developed system is limited by my capacity to debug, observe and understand the problem to make the contribution work.

11.1 Summary of our Contributions

This section summarizes our main contributions. Some of them have production maturity, and others do not. These contributions are all running on or for targeting the Kalray MPPA® processor. Most of these contributions could be generalized to any clustered manycore architectures.

In Chapter 5, a new communication technique and Application Programming Interface (API) has been designed and implemented at bare-hypervised level to reach high-throughput and low-latency. In the contribution, we explain in details the runtime initialization and the allocation of the hardware Direct Memory Access (DMA) Network on Chip

(NoC) resources. The distributed communication runtime provides asynchronous one-sided communication with a relaxed memory consistency model at the multi-cluster level. The runtime initialization is quite complicated, but the execution part has been refactorized many times to converge to a high-performance and light-weight implementation. Results show nearly peak hardware throughput when using data transfers of size greater than 4 kilobytes. On the Kalray 3rd MPPA® generation, part of the proposed software engines are performed in hardware, making the Input/Output Operation per Second (IOPS) throughput almost 20 times better than our software implementation. Indeed, this contribution helped both the hardware functional specification and the hardware implementation specification.

In Chapter 6, a new highly efficient multi-threading runtime is proposed. The runtime is implemented in the Kalray bare-metal toolchain right over the Kalray's exokernel. The runtime enables POSIX threads and OpenMP multi-threading to make efficient fine-grained threading possible. Results show a reduction of the execution times of up to 22 compared to the state-of-the-art. Such improvements are made possible thanks to the use of lock-free implementations, at the cost of complexity. The contribution has a high maturity level as it is used in many Kalray's products as the main multi-threading runtime running in the Compute Clusters (CCs) and Input/Output Subsystems (IOs) for the MPPA® processor. Moreover, this runtime also passes the entire C/C++/Fortran test-suite of the libgomp of the GNU Compiler Collection (GCC) project.

Chapter 7 presents a strategy to statically map an academic dataflow programming model efficiently on a manycore architecture with multiple levels of parallelism. Designed and implemented in the Institute of Electronics and Telecommunications of Rennes (IETR), the Parallel and Real-time Embedded Executives Scheduling Method (PREESM) framework uses a dataflow graph compilation technique that systematically flattens all graph hierarchy in the first compilation pass. This process often ends up in producing many actors to be mapped on cores, which is problematic on manycore architecture as the optimal mapping and scheduling problem is known to be NP-complete. Therefore, we proposed in this thesis a technique that consists in keeping/exploiting the graph hierarchy, and performing only a hierarchical mapping at coarse-grained. The fine-grained parallelism, inside the hierarchy, is retrieved by generating parallel for-loops using OpenMP multi-threading. Such a technique reduces the mapping complexity and preserves application parallelism. Experimental results show a mapping time reduction of more than 1000, and 6 % better performances than the systematic flattening.

Chapter 8 presents an embedded reconfigurable dynamic dataflow runtime for a clustered manycore architecture. The original runtime, designed and implemented by the Video Analysis and Architecture Design for Embedded Resources (VAADER) team at IETR, was introduced in 2014 [HPD+14]. As clustered manycore architectures feature distributed local memories with a partitioned global address space, this embedded reconfigurable dataflow runtime had to be deeply modified to make explicit inter-process communications possible (automatically). The master runtime, that schedules and distributes the work of the dataflow application, is statically mapped onto a host multi-core Central Processing Unit (CPU) (IO) and the workers are placed on the matrix of CCs. Each core of the MPPA® processor triggers the inter-process communications automatically when jobs are sent to them by the host runtime. Among the main contributions, the scheduler and the memory allocator have been redesigned. A light-weight scheduling heuristic is introduced to let the host keep up with the high degree of parallelism of manycores, and to provide efficient use of the on-chip local memories. Finally, as the local memories are small with

1.5 megabytes at most, a parallel deadlock avoidance algorithm is added to avoid local memory starvation and for sharing the memory between the cores inside one CC.

In Chapter 9, a new distributed framework for executing low-latency OpenVX applications on a clustered manycore processor is proposed. OpenVX is a modern and standard API that is defined by the Khronos Group for describing computer vision and inference neural network applications. OpenVX is built as an acceleration programming API where the computation is described by a Directed Acyclic Graph (DAG). As Directed Acyclic Graphs (DAGs) are used to express the computation, and a verification pass has to be issued before the actual processing of the OpenVX graph, it is possible to perform many optimizations during this verification. Our framework is an embedded runtime, that operates in user-space, running on the manycore processor host. The framework does not require any external resources or offline analysis to operate. It runs standalone. On the MPPA® processor, the front-end OpenVX framework is mapped on a host (IO) which deals with the OpenVX buffers and graphs described in the application by the developer. The tasks are offloaded on the acceleration matrix of CCs. Therefore, if the OpenVX graph changes, or if an external event changes a parameter, the user can retrigger the graph verification. It will reoptimize the overall execution of the computing graph pipeline automatically. To minimize the execution time, each kernel is automatically distributed on all available CCs of MPPA®. In the proposed framework, it is possible to specify at the creation of the OpenVX context a list of CCs, the number of cores in each CC, and to disable optimization passes. By default, this distributed framework performs automatic scheduling and static memory allocation of the graph execution, and fully automated optimization such as automatic and explicit Remote Direct Memory Access (RDMA) pre-fetching and kernel fusion. Pre-fetching and kernel fusion are the keys of performance on manycore machines as they respectively enable the overlapping of the computations and the communications, and avoid the main memory bandwidth wall which is one of the most significant problems in high-performance implementations. Such automated optimizations allow the user to reach super-linear speedups at the multi-clusters level, showing that the manycore architecture is efficiently exploited.

In Chapter 10, new distributed low-level implementations of diverse applications are described: a 3D stencil application, a Fast Fourier Transform (FFT), and a custom runtime for low-latency execution of inference Convolutional Neural Network (CNN) applications. The parallelization strategy and mapping of each use-case is explained as well as its implementation. Classical hand-written optimizations are used to reach competitive performances. At the core level, a manual pipeline of streaming memory accesses is used to provide efficient sparse memory accesses. Single Instruction, Multiple Data (SIMD) instructions are also exploited. At the multi-core level, explicit Pthread and OpenMP are then used for parallelizing each kernel inside each CC. At the multi-cluster level, distributed and asynchronous one-sided communications are leveraged to both pre-fetch data from the main memory (hide the high external memory access latency) and perform inter-cluster communication to avoid the main memory bandwidth wall. All optimizations described in this chapter are done manually. Experimental results show linear speedups when the main memory bandwidth no longer bounds the performances.

11.2 Future work

Some contributions listed below of this thesis deserve to be enhanced either by adding functionalities, or removing current limitations, or entirely re-designed due to the lack of

time and for proof-of-concept work. Others will need more engineering efforts to reach a higher maturity, required for production software.

11.2.1 Fundamental Mechanisms for Programming Manycores: Asynchronous One-Sided (AOS)

Relaxed Remote Atomic Operations

Currently, remote atomic operations are strictly ordered with respect to RDMA operations on a memory segment. Such a feature is useful for programmers and makes data transfers and synchronization transactions efficient on distributed local memory architectures. The programmer performs locally posted asynchronous operations, and the core can switch immediately to the next computation. The ordering is ensured by the runtime proposed in Chapter 5.

In some cases, such an ordering is not useful. For instance, when implementing end-to-end software flow-control mechanisms or when trying to synchronize several initiators with each other as fast as possible, the ordering feature can result in a significant software overhead. For this purpose, the solution is to provide relaxed (unordered) remote atomic operations. The ordering could be requested by a specified flag when calling the remote atomic operation. Such a feature is quite complicated to implement. It will require more hardware resources on the DMA NoC interface of MPPA® to handle the completion and the remote notification.

Efficient Collective Operations, with Broadcast

Efficient collective operations (like barrier or broadcast operations) are complex to design and implement on distributed memory architectures. Collective operations define concurrent communications between several CPUs, usually in a distributed environment. Collective operations are usually state-full. Each participant of the distributed communication has to be aware of the states of all participants in the communication.

For instance, collective operations ease the implementation of global barriers and broadcast operations. Broadcast operations are essential to reduce the pressure on the main memory, but it depends on how they are implemented. For instance in this thesis, in the case of the targeted manycore architecture, data can be read only once, and all participants receive them. However, collectives are challenging to implement because of resources sharing, lock-free mechanisms to make it efficient, and the high concurrency.

The support of collectives over the NoC of the MPPA® processor will be a significant advance, in the software stack for developing and optimizing parallel distributed applications.

Large Scale Asynchronous One-Sided: Multi-MPPA Support

Asynchronous one-sided communications are only supported on a single MPPA® chip. The current implementation has not been designed to scale with multiple MPPA® chips. However, redesigning the way the hardware resources are shared would make the support of asynchronous one-sided communications possible over multiple MPPA® chips. As the implementation will be even more concurrent, such distributed software development will require time and careful design. Moreover, the debugging of such implementation is very complex and will require more instrumentations to let the designer understand what happens.

Asynchronous One-Sided with Kernel Bypass onto Linux

Generic and robust Linux drivers are complicated and have significant overheads. Therefore, when optimizing the execution time of the communications of a Linux application, the Linux driver is often the bottleneck. State-of-the-art optimizations use kernel bypass techniques to access the DMAs from the user-space directly.

Such optimizations are complex and dangerous, but they provide the user with low overhead implementations. Indeed, kernel bypass techniques imply many constraints such as the sharing of resources with other Linux processes and the driver itself, but also careful management of the memory regarding both the Linux virtual memory and the memory map of the peripherals.

11.2.2 Standard Optimized Runtimes for Manycores

Optimization of the Standard GCC OpenMP Runtime Library

The libgomp runtime provided by the GCC project is a generic implementation of the OpenMP runtime. It uses the POSIX multi-threading backend. Therefore, the multi-threading runtime presented in Chapter 6 optimizes the implementation of the POSIX thread primitives. However, the standard OpenMP runtime of GCC is not. For instance, the OpenMP runtime of GCC does not implement lock-free mechanisms to update shared variables. This runtime also uses intensively the dynamic memory allocator which is quite slow.

The first contribution could be the bypassing of some primitives of the POSIX threads used inside the GCC OpenMP runtime. Directly mapped atomic instructions could replace these primitives. Moreover, atomic instructions could be used to update concurrently shared variables inside the OpenMP runtime of GCC. Finally, an important optimization could be the static memory allocation of internal memory resources of the OpenMP runtime. Indeed, the dynamic memory allocations performed by the OpenMP runtime has a significant overhead on fine-grained multi-threading.

OpenMP 4.0 Support for MPPA®

OpenMP has been widely used in both rapid parallel implementations and high-performance parallel implementations. Moreover, OpenMP is very appreciated for production software as the parallelization only consists in inserted few compilation directives in the original code. Indeed, in production software, the modification of the code costs a lot. As such OpenMP is well suited to parallelize applications efficiently at a reduced cost.

With a modern GCC or Low-Level Virtual Machine (LLVM) compiler, it is possible to support OpenMP 4.0. It will be a significant advance in the software stack for offloading computation from a Linux or a Real-Time Operating System (RTOS) to the acceleration matrix of CCs.

A High Efficient Light-Weight OpenCL for MPPA®

The current OpenCL support of MPPA® uses an open-source front-end that requires a Linux system. In some embedded systems and high-performance designs, Linux is banished. Therefore, the need for a light-weight and high efficient (Linux independent) OpenCL support is essential.

The idea is to build on top of the bare-metal distributed offloading runtime, presented in Section 9.2, a new OpenCL front-end. The OpenCL runtime and compiler front-end

should support the data parallel and the task parallel modes. The two modes could switch at runtime, depending on use-cases. However, the host would have to compile the OpenCL kernels off-line. Otherwise, the IO of MPPA® should run a compiler. With such a feature, efficient and fine-grained offloading of computations from the IO of MPPA® to the CCs could be done. It should ease the programming of MPPA® for application engineers.

Moreover, standard extensions, like the one presented in Section 5.6.2, have to be supported to provide software engineers with optimization tools.

A vendor specific feature for efficient exploitation of the distributed local memories of the MPPA® processor has to be available: the exposition of inter-CCs communications in the compute kernel of OpenCL at runtime.

11.2.3 Parallelization Techniques

Off-chip RDMA-based Time-skewing

Time-skewing techniques reduce the main memory bandwidth by restructuring memory accesses of an application to increase data locality in iterative stencil applications. Many works, such as polyhedral optimizations use and exploit time-skewing techniques, but they are all based on *Load/Store* memory accesses. Such optimizations could also be applied to dataflow programming in a tool like PREESM or an embedded dataflow runtime like Synchronous Parameterized Interfaced Dataflow Embedded Runtime (SPIDER).

Automatic off-chip RDMA-based time-skewing has never been attempted. Such a contribution could break the current limitation of kernel fusion optimizations that are used in the proposed OpenVX low-latency implementation (see Chapter 9).

The idea is to write an explicit tiling algorithm which performs automatic time-skewing. The algorithm is given a number of steps which should be computed, before spilling the entire data-set in the main memory (external).

Such an algorithm is complex to design as it implements multi-dimensional memory access patterns, asynchronous inter-CCs communications in a highly concurrent environment, and it must be flexible enough to execute various computations at the different skewed steps.

Toward a Warp Inspired Execution Model

On Graphics Processing Units (GPUs), a warp is a parallel code section that is executed temporally in parallel. All cores that are participating in the execution of a warp execute the same instruction at each clock cycle (Single Instruction, Multiple Threads (SIMT)). As there is hardware multi-threading onto GPUs, when a warp accesses to a location of data in the memory hierarchy, the stall on the Read-After-Write (RAW) dependency make the entire warp switch to another warp in the clock cycle. Therefore, thousands of threads are used to overlap computations and communications by *Load/Store* on GPUs thanks to this hardware multi-threading.

On MPPA®, such hardware multi-threading is not possible but a warp inspired execution model could be made possible at a higher level of granularity. The (hardware) multi-threading of GPUs is done on *Load/Store* transactions, whereas the (software) multi-threading on MPPA® could be done on DMA transactions. The idea would consist in starting several structured teams (groups) of POSIX threads (it could be OpenMP teams), and the master thread of each team performs DMA transactions with the main memory. On the completion of the DMA transaction, the entire team of threads is rescheduled by the multi-threading runtime. Such a contribution is quite challenging to design as it involves

low-level runtime implementation, asynchronous communications, and updates of shared variables using lock-free mechanisms for performance.

11.3 Final Conclusion

This thesis gave me the opportunity to work on software runtimes for helping with the programming of a new generation of clustered manycore architectures. Throughout this work, I proposed, developed, and validated fundamental low-level mechanisms for programming new clustered manycore architectures with local memories. The complexity of these architectures should be hidden by programming models allowing the application engineers to optimize their applications. However, exotic programming models are complicated to be pushed into mainstream programming due to software legacy and portability. I believe that standard, well-documented, and high-level APIs (like OpenMP, OpenVX, Vulkan, and new programming models) are a competitive answer to abstract these architectures. As vendors usually implement these APIs, they likely provide application engineers with the best possible performance of the platform. One of the biggest challenges is to make these high-level programming models and tools (like compilers) able to reach peak performance of the platform, to make the life of application engineers easier, and to provide a faster time to market.

ANNEXE A

French Summary

Le monde de l'informatique est vaste et il a changé nos modes de vie depuis maintenant des décennies. Depuis l'invention du transistor dans les années 50 jusqu'aux tous nouveaux téléphones intelligents, voitures autonomes et maisons intelligentes, le monde que nous connaissons aujourd'hui est devenu dépendant des systèmes informatiques. Ces exemples font partie de ce qui est couramment appelés les "systèmes informatiques embarqués". Ces systèmes visent à remplir des fonctions spécifiques dans un environnement restreint avec diverses contraintes telles que l'énergie, le coût, la place, la durée de vie et les performances. Ajouté à cela, depuis l'émergence des machines de calcul et de l'informatique, le besoin en calcul n'a jamais cessé de grandir. En terme de puissance de calcul, la demande des applications modernes est de plus en plus énorme. L'automatisation, la rapidité d'exécution et les services fournis permettent plus de facilité, souplesse et économie de diverses manières.

Cependant l'informatique (embarquée) moderne est de plus en plus difficile à appréhender pour les ingénieurs et les utilisateurs des systèmes informatiques. C'est d'autant plus vrai pour les ingénieurs amenés à programmer la machine à niveau relativement bas (assez proche de la machine). Les systèmes informatiques modernes sont très concurrentiels et hétérogènes ce qui rend leur utilisation difficile. Pour palier à ce problème, des méthodes et modèles de calcul existent afin de réduire cette complexité et abstraire le plus possible l'architecture de la machine de calcul utilisée. Cela est encore plus vrai en ce qui concerne les systèmes informatiques parallèles et embarqués qui seront au centre des travaux présentés.

Dans cette thèse, des modèles de calcul tels que les modèles flux de données sont utilisés, mais aussi des interfaces de programmation d'applications sont exploitées et implémentées en partant de rien, de la machine à nu (le métal). Grâce à elles, il est possible de réduire considérablement les temps de développement des logiciels informatiques déployés sur des machines parallèles complexes. Comme les applications embarquées parallèles tendent à devenir dynamiques mais toujours avec des parties statiques, et avec de fortes contraintes en termes de performance, à savoir en cadence et latence, il est nécessaire que l'outillage servant au développement de ces applications soit robuste, efficace, utilisable (le plus facilement possible pour l'ingénieur non expert de la machine), observable et débogable.

A.1 Systèmes parallèles embarqués

Un système embarqué est n'importe quel système informatique qui n'est pas destiné à être un système informatique à tout faire. Le système embarqué est aussi associé au calcul informatique avec contraintes qui peuvent parfois être très agressives en fonction des applications.

Le parallélisme, souvent assimilé à la concurrence de l'exécution du système informatique, est l'exécution simultanée de plusieurs opérations sur des ressources matérielles et/ou logicielles différentes. L'exécution concurrentielle peut être observée à différents niveaux de granularité. L'opération vectorielle permet en une seule instruction machine d'effectuer la même opération sur plusieurs données en parallèle au niveau d'un cœur. A ce niveau, il est aussi trouvé la capacité d'exécuter plusieurs instructions par cycle par cœur. Ces cœurs sont dits super scalaire (ordonnancement dynamique) ou encore Very Long Instruction Word (VLIW) (ordonnancement statique). Un autre niveau de parallélisme est celui de l'exécution multitâche où plusieurs cœurs (multi-cœurs) de calcul accédant à une mémoire partagée peuvent réaliser des calculs de manière concurrentielle avec leur propre contexte d'exécution, indépendamment les uns des autres. Le dernier niveau de parallélisme, situé à une granularité plus élevée, est le multi-nœuds. Un nœud de calcul est un multi-cœur qui implémente plusieurs tâches fonctionnant aussi de manière concurrentielle. Au niveau logiciel, avec un tel niveau de parallélisme, la programmation est aussi appelée "programmation distribuée". Elle est souvent difficile à mettre en œuvre notamment à cause des communications et synchronisations devant être mises en place.

A.1.1 Le parallélisme et le processeur MPPA® de Kalray

Dans cette thèse, le processeur Multi-Purpose Processor Array (MPPA)® de Kalray est la cible principale pour l'évaluation des différentes contributions. Kalray est une entreprise innovante Française de semi-conducteur. Elle est pionnière dans la conception de processeurs massivement parallèles et de ses méthodes de programmation. Le MPPA® permet d'exploiter tous les niveaux de parallélisation qui viennent d'être décrits. La Figure A.1 montre intuitivement la hiérarchie des différents niveaux de parallélisme qui peuvent être exploités sur le processeur MPPA® de Kalray par les couches logicielles et applicatives.

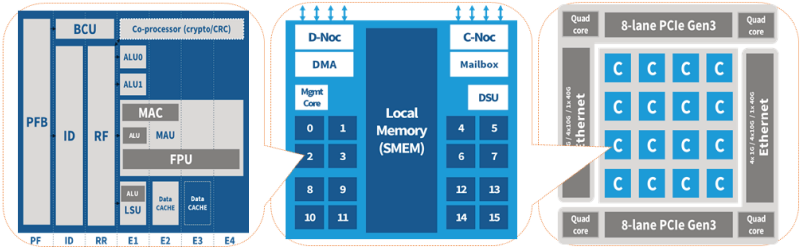


Figure A.1 – *Le processeur MPPA® de Kalray*

A bas niveau, le MPPA® offre un cœur de calcul VLIW capable d'exécuter jusqu'à 5 instructions par cycle, dont certaines instructions pouvent elles-mêmes travailler sur des vecteurs de registres en parallèle. Ensuite ces cœurs sont regroupés dans un nœud multi-cœurs de 16 cœurs applicatifs avec un cœur supplémentaire de management du système, soit 17 cœurs. Le processeur MPPA® de Kalray intègre jusqu'à un total de 16 de ces nœuds multi-cœurs de 17 cœurs, ce qui le rend extrêmement concurrentiel et difficile à programmer. La mémoire interne de ces machines multi-cœurs est une mémoire locale et non un cache

comme il est classiquement trouvé sur les processeurs parallèles. Les mémoires locales du processeur MPPA® impliquent une gestion explicite par logiciel de la communication entre la/les mémoire(s) externe(s) et interne(s) de la machine. Ces mémoires sont toutes inter connectées par un réseau sur puce capable de véhiculer jusqu'à 4 octets par cycle dans les deux sens. Cette gestion explicite de la communication est un vrai défi pour les ingénieurs logiciels. Cependant une telle caractéristique rend le processeur MPPA® très efficace en terme de rapport de puissance de calcul et d'énergie, mais également stable concernant ses temps de calcul pour une application donnée.

A.1.2 Mémoires et protocoles de communication

Dans les systèmes embarqués, la mémoire est une ressource précieuse. Même si la plupart de la surface en silicium d'un processeur est de la mémoire (en moyenne 80%), celle-ci doit être gérée et utilisée avec précaution. Il existe différentes mémoires dans les systèmes informatiques modernes. La file de registres est une petite mémoire dans les cœurs de calcul où l'exécution courante se passe. Les registres ont une très faible latence d'accès qui est de l'ordre du cycle machine. Il est ensuite trouvé différents niveaux de mémoire sur puce qui servent à éponger et réduire les attentes de réponse du système de mémoire externe. En effet, les mémoires externes mettent un temps élevé à répondre, comparé à la fréquence de fonctionnement des processeurs. Dans cette thèse, une partie importante des travaux consiste à utiliser efficacement le système mémoire sur puce avec notamment l'utilisation de communications explicites ou par le pré-chargement de données au travers de la hiérarchie de cache.

Comme les communications explicites sont incontournables pour programmer le processeur MPPA® de Kalray et plus généralement les machines implémentant des mémoires locales, deux protocoles de communication fondamentaux sont mis en avant. La première est la communication bilatérale. Le MPPA® expose un réseau sur puce qui permet de communiquer seulement de manière bilatérale. C'est à dire que la communication peut être initiée seulement si le receveur et le transmetteur sont tous les deux prêts : données consistantes, ressources matérielles configurées, et synchronisées. La communication bilatérale a donc une caractéristique de correspondance très stricte entre l'émetteur et le receveur. Cela peut parfois être handicapant concernant l'optimisation de la communication notamment sur l'ordre. La seconde est la communication unilatérale qui permet à l'initiateur de la communication d'être maître sur une ou plusieurs mémoires distantes. La communication unilatérale permet la relaxation du système mémoire, c'est à dire que le système mémoire offre la capacité d'ordonnancer les transactions mémoires dans le désordre afin d'être plus performant. L'ordre et les synchronisations avec le système mémoire sont assurés par des opérations dédiées à cet effet, faites a posteriori. Ces opérations ont pour but de maintenir la consistance mémoire entre plusieurs initiateurs. Typiquement, garantir que les écritures et lectures d'un cœur et d'un périphérique en mémoire seront correctement ordonnées, afin que l'un, puisse voir les écritures de l'autre dans une mémoire (partagée ou distribuée).

A.2 Les modèles de programmation parallèle

Les modèles de programmation parallèle permettent d'abstraire plus ou moins la machine sur laquelle s'exécute le programme. Certains modèles sont relativement proches du matériel alors que d'autres sont de haut niveau. Très souvent, plus le modèle de programmation est loin de la machine d'exécution, moins l'utilisateur a le contrôle sur ce qui s'exécute sur

la machine. Les travaux présentés mettent l'accent sur des modèles multitâches et modèles d'accélérations d'applications.

A.2.1 Interfaces de programmation d'applications

Les interfaces de programmation d'applications sont aujourd'hui très réputées car elles permettent d'exploiter les caractéristiques matérielles et logicielles exposées par les vendeurs ou développeurs logiciels de la machine visée. L'interface de programmation d'applications Pthread et OpenMP sont standard et permettent d'exploiter le parallélisme sur un modèle de mémoire partagée et symétrique. Ce modèle a pour but d'exploiter efficacement une machine parallèle et symétrique. Il donne un contrôle assez fin sur les cœurs de calcul, ce qui permet de contrôler à bas niveau le cycle de vie des tâches et leurs ressources. Les environnements d'exécution multitâches utilisent principalement des mécanismes d'exclusions mutuelles (verrous) afin de gérer le partage de ressources entre les différents cœurs de calcul. OpenCL, OpenMP4 et Vulkan sont des interfaces de programmation standard qui offrent la possibilité de déployer un calcul ou une portion de calcul (lourde) sur un accélérateur. Le but est de décharger la machine sur laquelle tourne l'application principale d'un calcul gourmand en ressources (calcul et mémoire) pour l'accélérer sur une ou plusieurs ressources de calcul externes. OpenCL et Vulkan sont globalement assez proches de la machine visée car ils exposent des caractéristiques spécifiques à l'architecture telles que la hiérarchie mémoire, le nombre de cœurs, la taille des mémoires locales, et des accélérations spécifiques au calcul visé.

A.2.2 Modèles de flux de données

Les modèles de flux de données sont très intéressants pour la description d'applications qui sont contrôlées par le mouvement des données. Le principe de base d'un modèle de flux de données correspond à un graphe où les nœuds représentent des calculs sur les données d'entrée et produisent une ou plusieurs sorties. Les arcs représentent les connections entre ces nœuds qui décrivent donc les dépendances du calcul à réaliser. Beaucoup de modèles de flux de données existent, mais la plupart sont des modèles non standard. Dans cette thèse, des modèles de flux de données, développés par Institute of Electronics and Telecommunications of Rennes (IETR) dans l'équipe Video Analysis and Architecture Design for Embedded Resources (VAADER) seront exploités afin d'utiliser le plus efficacement possible une architecture massivement parallèle. Le premier modèle est un flux de données hiérarchique et statique qui permet un réglage fin des différents niveaux de parallélisme possibles dans les architectures modernes. Le second modèle de flux de données réconcilie les modèles de flux de données dynamiques non déterministes et le pleinement statique. Appelé Parameterized and Interfaced dataflow Meta-Model (PiMM), ce modèle compositionable donne la possibilité de reconfiguration dynamique des graphes composant l'application. Cette reconfiguration peut être effectuée une seule fois par itération de graphe, ce qui donne au modèle plus de prédictibilité.

A.3 Environnement d'exécution bas niveau pour architectures massivement parallèles

Cette section décrit les deux principaux environnements d'exécution bas niveau proposés dans cette thèse et qui seront ensuite utilisés dans toutes les autres contributions de

cette thèse, mais aussi dans la plupart des produits logiciels fonctionnant sur le processeur MPPA® deuxième génération de Kalray à partir de 2017.

A.3.1 Environnement distribué pour la communication asynchrone unilatérale

Les principaux travaux de recherche sur la communication unilatérale (one-sided communications en Anglais) sont principalement liés à la communication visant à échanger des données entre des nœuds de calcul qui ont d'énormes mémoires, soit plusieurs gigaoctets de mémoire. L'état de l'art de ces systèmes de communication résulte donc à des propositions d'interface de la communication de plusieurs mégaoctets. Cela n'est pas acceptable pour une architecture massivement parallèle avec des mémoires locales de 2 mégaoctets. Cette contribution, qui est la principale brique sur laquelle repose les autres contributions de cette thèse, permet de faire de la communication asynchrone très efficace sur le processeur MPPA® qui implémente un réseau connecté de mémoires locales distribuées.

Les principaux défis de cette contribution sont la gestion du partage de ressources matérielles, l'asynchronisme de l'implémentation et le fait que l'environnement distribué de communication soit hautement concurrentiel. En effet, l'émulation de communications asynchrones unilatérales sur un réseau du puce capable de communiquer seulement de manière bilatérale implique une forte assistance logicielle. Cette assistance logicielle met en œuvre de l'ordonnancement de tâches de transferts, des mécanismes de contrôle de flux basés sur des crédits logiciels, et partage des données de contrôle sans mécanisme d'exclusion mutuelle. L'absence de mécanisme d'exclusion mutuelle permet d'atteindre une haute performance sur le logiciel multitâche grâce à l'utilisation des opérations atomiques du cœur et des périphériques.

Cet environnement expose la machine MPPA® comme un réseau de mémoires où n'importe quel nœud multi-cœurs peut avoir accès à n'importe quelles mémoires au travers du réseau sur puce. Cette communication peut être initiée de manière asynchrone afin de pouvoir masquer les latences de communication avec du calcul. L'accès à la mémoire distante est possible en ouvrant une fenêtre mémoire contiguë sur la mémoire locale. Cette fenêtre peut ensuite être clonée pour faire le lien entre les deux participants à la communication, qui est illustrée sur la Figure A.2. Cette action est synchronisante, ce qui rend l'utilisation de l'environnement de communication plus simple vis à vis des éventuelles situations de course à l'initialisation des applications distribuées sur le processeur MPPA®. Une fois que le canal de communication unilatéral est établi, l'utilisateur peut initier des communications asynchrones fournissant une haute bande passante en fonction de l'état global du système (contention du réseau sur puce, de l'initiateur ou du serveur). L'environnement distribué de communication permet aussi de synchroniser efficacement différents nœuds multi-cœurs grâce à l'émulation logicielle d'opérations atomiques dans les mémoires locales distantes des nœuds. Ces opérations ont la propriété d'être strictement ordonnées avec les transactions mémoires asynchrones en vol sur la fenêtre mémoire distante visée. Cela permet à l'initiateur de poster toutes les opérations localement sans avoir à attendre les complétions de celles-ci. L'ordre est géré automatiquement à distance par l'environnement de communication proposé dans cette contribution.

L'implémentation logicielle étant dans le chemin critique de la performance, l'environnement permet d'atteindre 70% de la performance crête du processeur pour des transferts mémoires supérieurs ou égaux à 8 kilo octets. Sur la prochaine génération du processeur MPPA®, certaines parties du logiciel de cet environnement seront matérialisées pour atteindre la crête du processeur sur des petites transactions mémoires.

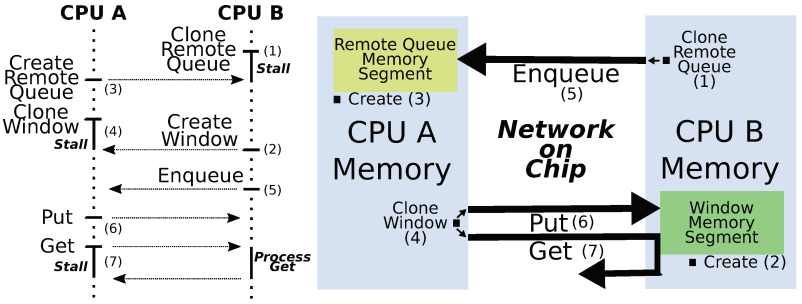


Figure A.2 – *Utilisation des segments mémoires et des protocoles*

A.3.2 Environnement d'exécution multitâche symétrique performant : sans verrou et transactionnel

La parallélisation efficace dans les nœuds multi-cœurs d'un processeur massivement parallèle tel que le MPPA® est fondamentale. Cette contribution propose et optimise un environnement d'exécution dans la chaîne d'outils métal à nu du MPPA® de Kalray qui est basée sur le compilateur GNU Compiler Collection (GCC). Cet environnement permet de gérer efficacement à bas niveau les tâches exécutées en parallèle dans les nœuds multi-cœurs du MPPA®. L'environnement propose un sous ensemble de primitives du standard Pthread et les optimise pour une exécution à basse latence. Les principales fonctionnalités sont la gestion du cycle de vie des tâches, les mécanismes d'exclusions mutuelles, les sémaphores et les barrières multi-cœurs. Tous ces mécanismes sont élaborés sans verrou (pas de mécanismes d'exclusions mutuelles) à l'intérieur de l'environnement afin d'être très efficace. Cela est possible grâce à l'utilisation élégante des instructions atomiques des cœurs et des techniques de mémoire transactionnelle logicielle.

La première étape consiste à activer le multitâche dans la chaîne d'outils métal à nu du processeur MPPA®, et passe donc par l'activation de la définition standard des différentes fonctions Pthread. Cela est réalisé au moment de la construction de la *newlib* qui est une librairie de fonctions relativement bas niveau qui sont utilisées classiquement dans le langage C (`printf`, `malloc` etc). Grâce à cela, il est ensuite possible d'activer la construction de l'environnement d'exécution OpenMP. OpenMP est une interface de programmation qui permet de paralléliser en quelques lignes de directives de compilations, une application séquentielle. Ces lignes sont ignorées si le compilateur ou l'environnement d'exécution ne les supportent pas. Grâce à l'environnement proposé et au portage de GCC pour l'architecture MPPA®, l'OpenMP peut être utilisé très efficacement pour paralléliser dans un nœud multi-cœurs une application. L'environnement proposé et optimisé est aujourd'hui utilisé dans la plupart des produits et applications développés par les ingénieurs sur MPPA®. L'environnement a une haute maturité car il est capable de faire fonctionner toute la validation standard C/C++/Fortran OpenMP de GCC. De plus, les résultats montrent une réduction de latence mesurée jusqu'à fois 18 sur la machine MPPA® seconde génération comparé à l'environnement d'exécution original.

A.4 Exécution d'applications de flux de données pour architectures massivement parallèles

Une nouvelle stratégie pour exécuter un modèle de flux de données hiérarchique et statique sur une architecture massivement parallèle est présentée dans cette section. Il est aussi

proposé une adaptation d'un environnement d'exécution embarqué pour ordonnancer à l'exécution un modèle de flux de données paramétré et reconfigurable.

A.4.1 Stratégie pour ordonnancer efficacement un modèle statique de flux de données hiérarchiques

Le placement et l'ordonnancement de tâches sur une architecture massivement parallèle n'est pas trivial. Il a été démontré que le placement et l'ordonnancement est un problème NP-complet. Cela signifie donc que le problème ne peut pas être résolu de manière optimale en un temps polynomial. En effet le temps de placement et ordonnancement augmente de manière exponentielle avec le nombre de cœurs et le nombre de tâches à placer sur ces cœurs de calcul. Basée sur un modèle de flux de données hiérarchique et statique, la stratégie pour placer et ordonnancer une application parallèle exploite les deux niveaux de hiérarchie de parallélisme de la plateforme et la hiérarchie du modèle de calcul. Le modèle flux de données hiérarchique et statique implémente des acteurs qui sont des calculs atomiques et indivisibles. Ces acteurs sont inter-connectés par des arcs qui contiennent des données. La hiérarchie du modèle permet d'associer un acteur à un sous graphe flux de données.

La preuve de concept est évaluée dans Parallel and Real-time Embedded Executives Scheduling Method (PREESM) qui est un outil graphique qui permet de décrire des applications de flux de données conçu par l'équipe VAADER au sein du laboratoire IETR. Originalement, le processus de compilation d'un graphe de flux de données hiérarchique applicatif était systématiquement mis à plat. La mise à plat d'un graphe flux de données hiérarchique consiste à remplacer les acteurs hiérarchiques par leurs graphes correspondants. Cette mise à plat peut produire beaucoup d'acteurs qui devront ensuite être placés et ordonnancés sur les cœurs de la plateforme. Dans cette contribution, la mise à plat n'est plus nécessaire (mais toujours possible) ce qui permet d'avoir moins d'acteurs à placer et à ordonnancer sur la plateforme visée. Cela provoque une perte de parallélisme, mais celui-ci est ensuite exploité par la génération de boucle à itération finie et parallèle. Le groupage d'acteurs permet de construire cette représentation, et un ordonnancement d'acteurs à exécuter de manière séquentielle et itérative. Cette représentation est ensuite récupérée au moment de la code génération pour l'architecture visée. Des directives de compilation OpenMP sont automatiquement générées afin de récupérer le parallélisme perdu en amont. Les résultats montrent une immense réduction du temps de placement et ordonnancement en utilisant cette méthode, ainsi qu'un gain en performance qui est du à la fusion de certains accès à la mémoire principale. Cette fusion est apportée par la passe de groupage d'acteurs dans les sous graphes qui n'ont pas été mis à plat.

A.4.2 Portage et adaptation d'un environnement d'exécution embarqué pour placer et ordonnancer un modèle flux de données paramétré et dynamique

Les systèmes embarqués complexes qui utilisent des modèles de flux de données tendent de plus en plus vers la reconfigurabilité dynamique. Le modèle de flux de données permet d'exploiter la reconfiguration partielle ou totale de l'application, et donc de son parallélisme interne. Appelé Synchronous Parameterized Interfaced Dataflow Embedded Runtime (SPIDER) et proposé par l'équipe VAADER dans le laboratoire IETR, cet environnement d'exécution embarqué permet de placer et ordonnancer à l'exécution un méta modèle de flux de données paramétrique sur une plateforme parallèle. L'environnement a été initialement implémenté pour supporter les architectures parallèles symétriques à mémoire partagée. SPIDER adopte une approche maître et travailleur où l'ordonnanceur fonctionne

sur le maître et les travailleurs exécutent les commandes de calcul envoyées par le maître. L'environnement est proprement architecturé en langage C++ avec une partie plateforme spécifique, un ordonnanceur, un manageur de mémoires partagées dans les nœuds multi-cœurs, et des mécanismes abstraits de communications et synchronisations.

Dans cette contribution, le processeur MPPA® de Kalray implémentant un réseau de mémoire distribuée est visé. Le maître de SPIDER est placé sur un multi-cœurs ayant accès aux entrées/sorties et les travailleurs sont placés sur les cœurs des multi-cœurs de la matrice d'accélération. Comme SPIDER a été développé sur un modèle de mémoires partagées, l'environnement doit être modifié et adapté sur un modèle de mémoires distribuées. Le principal défi est la gestion explicite de la communication, qui sur l'architecture à mémoire partagée se fait automatiquement par le cœur au travers de son unité de chargement et déchargement de données, alors que sur MPPA®, elle doit se faire de manière explicite par des transferts Direct Memory Access (DMA) logiciels. Ces transferts explicites sont utilisés pour véhiculer les données de contrôle des queues de commandes, ainsi que pour transférer les données d'entrées et de sorties des acteurs placés par le maître sur les cœurs des nœuds multi-cœurs de la matrice d'accélération. De nouveaux placeur et ordonnanceur sont proposés afin d'optimiser la cadence d'envoi des commandes, mais aussi afin d'utiliser plus efficacement la mémoire locale des nœuds multi-cœurs. La mémoire locale est une ressource précieuse, donc l'heuristique de placement est modifiée pour prendre en compte le taux d'utilisation des mémoires locales dynamiquement à l'exécution. L'idée principale est de réduire la contention sur les demandes d'allocations de données qui permettent l'exécution de l'acteur dans le nœud multi-cœurs. Les résultats sont prometteurs car ils montrent des facteurs d'accélérations proches de la limite théorique donnée par la loi l'Amdahl.

A.5 Un environnement standard distribué pour la vision et applications sur architectures massivement parallèles

Un nouvel environnement distribué pour l'exécution d'applications au standard OpenVX est proposé dans ces travaux. L'environnement distribué donne la possibilité à un utilisateur de faire fonctionner des applications OpenVX qui sont automatiquement parallélisées sur un processeur massivement parallèle. Cette section présente aussi plusieurs applications implémentées à bas niveau. Les méthodes de parallélisation sont expliquées mais également la façon dont elles ont été adaptées à un processeur massivement parallèle qui embarque des mémoires locales.

A.5.1 Environnement embarqué distribué pour l'exécution d'applications OpenVX à basse latence

OpenVX est une interface de programmation moderne et standard proposée par Khronos qui permet de déployer des applications de vision par ordinateur ou de réseaux de neurones en inférence (exécution d'un réseau déjà entrainé) sur un ou plusieurs accélérateurs. OpenVX est donc une interface de programmation en accélération où l'application fonctionne sur un hôte et le calcul est déporté sur l'accélérateur. La principale caractéristique de l'OpenVX comparée aux autres interfaces de programmation telles que l'OpenCV, est que le calcul est décrit par un graphe acyclique dirigé. Ce graphe est ensuite explicitement vérifié par une fonction du standard, et le vrai calcul est ensuite lancé par une autre fonction depuis l'hôte embarquant l'application OpenVX.

L'environnement distribué permettant le support de l'OpenVX sur le processeur MPPA® de Kalray est une nouvelle implémentation au C99 en partant de la spécification de l'interface

de programmation du standard de Khronos. La seule dépendance est celle de la librairie C, son mécanisme d'allocation de zone mémoire dynamique ainsi que d'un éditeur de lien dynamique fonctionnant du côté de la matrice d'accélération massivement parallèle. Une caractéristique importante de l'environnement est qu'il est totalement autonome. En effet, l'environnement fonctionne sans assistance externe. Il a seulement besoin d'un hôte multi-cœurs où est placé l'environnement qui implémente l'interface de programmation OpenVX. Le calcul intensif est ensuite déployé automatiquement sur les différents nœuds multi-cœurs de la plateforme massivement parallèle visée.

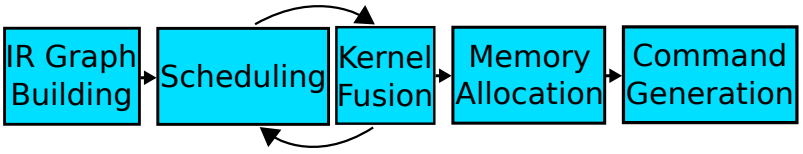


Figure A.3 – *Vérification et optimisation du graphe OpenVX applicatif - vxVerifyGraph [G+17]*

L'intelligence de cet environnement repose sur l'optimisation automatique de l'exécution de la chaine de calcul décrite par le graphe OpenVX acyclique dirigé. Cela signifie donc que l'environnement est capable de placer, ordonnancer et optimiser automatiquement les tâches de calcul depuis l'hôte multi-cœurs sur la matrice d'accélération, et tout cela est fait à l'exécution de l'application de manière autonome et embarquée. Le processus qui permet de faire cela est décrit par la Figure A.3.

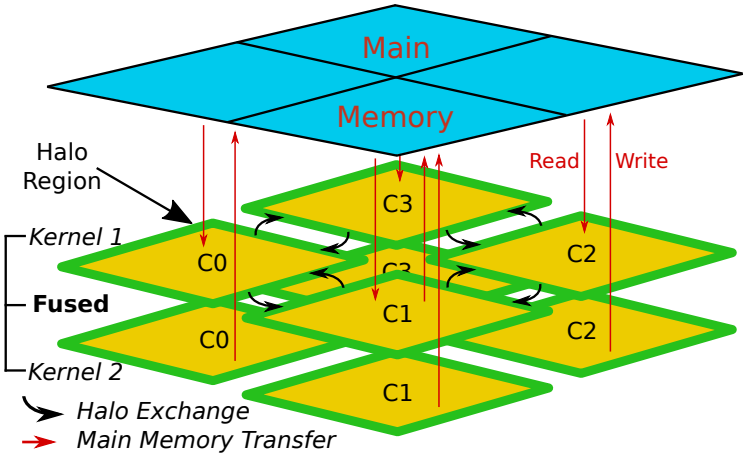


Figure A.4 – *Automatisation de la fusion de nœuds standard OpenVX à l'exécution*

Dans l'environnement, la représentation interne du graphe de calcul est d'abord construite à partir de la description standard OpenVX. L'ordonnancement est un tri topologique qui est raffiné avec une passe de fusion de nœuds de calcul afin d'éviter le plus possible les copies de données en mémoire principale. En effet la fusion de nœuds, illustrée par la Figure A.4, permet de sauver de la bande passante sur la mémoire externe qui est le principal bouchon de performance des applications sur les architectures massivement parallèles. Si la fusion n'est pas possible, un algorithme de tuilage automatique est appelé et celui-ci implémente du pré chargement automatique de tuiles afin de masquer les latences d'accès à la mémoire principale (masquer la communication avec du calcul automatiquement). Les causes d'impossibilité de faire de la fusion de nœuds sont par exemple les formes de tuiles non supportées, les échanges inter nœuds multi-cœurs très difficiles, ou encore le manque de mé-

moires locales adressable directement depuis les cœurs des nœuds multi-cœurs. Lorsque les optimisations sont finies, le processus de vérification du graphe OpenVX lance l'allocation statique des zones mémoires distribuées dans les mémoires locales des nœuds multi-cœurs de la matrice d'accélération. Ensuite, les commandes de calcul sont générées puis envoyées à la matrice d'accélération au moment du lancement de l'exécution du graphe fait par l'utilisateur OpenVX.

Puisque le besoin des applications embarquées (voitures autonomes par exemple) requière des systèmes les plus réactifs possibles, l'implémentation vise une exécution très basse latence. Pour répondre à ce besoin, chaque nœud du graphe OpenVX applicatif est distribué automatiquement sur toute la matrice d'accélération. Les résultats montrent des facteurs d'accélération supers linéaires au niveau multi-nœuds multi-cœurs, ce qui indique que la hiérarchie mémoire du processeur MPPA® de Kalray est bien utilisée.

A.5.2 Applications et environnements embarqués distribués à la main

Il est présenté plusieurs applications qui sont implémentées à bas niveau sur le processeur massivement parallèle dans cette section. Les travaux réinvestissent les environnements d'exécution proposés afin d'optimiser les applications implémentées à bas niveau sur la machine visée. Tout d'abord, chaque application est expliquée, caractérisée, et ses méthodes propres de parallélisation sont abordées. Trois applications sont visées. La première est une application de stencil 3D utilisée dans les simulations numériques pour prédire par exemple la météo ou encore simuler les phénomènes océanographiques. Ensuite, une implémentation distribuée de la transformée de Fourier est proposée. Celle-ci est fondamentale dans le traitement du signal. La dernière contribution est un environnement d'exécution très spécifique permettant de faire fonctionner des réseaux de neurones en inférence.

Les travaux permettent à l'utilisateur de déployer automatiquement l'application optimisée à bas niveau sur MPPA®. Les optimisations misent en œuvre sont basées sur les communications asynchrones unilatérales explicites et l'environnement multitâche proposé dans les nœuds multi-cœurs de la machine massivement parallèle. Les optimisations effectuées se concentrent sur le pré chargement de données, la restructuration des accès mémoires dans les mémoires externes afin d'augmenter la localité spatiale et temporelle. De plus, la fusion des différents processus de calcul est exploitée pour éviter le bouchon de bande passante des mémoires externes.

A.6 Conclusion

La programmation des processeurs embarqués parallèles et hétérogènes est un énorme défi. Les mémoires locales du processeur MPPA® font de lui une machine très efficace en termes de rapport de puissance de calcul et d'énergie. Cependant, la programmation d'une telle architecture est difficile pour les ingénieurs logiciels. Pour faciliter les développements et la maturation des applications logicielles sur ce type de processeur, différents outils ont été proposés et adaptés à l'architecture MPPA®.

Dans cette thèse, plusieurs approches sont exploitées. Premièrement, le problème de la communication explicite très efficace et relativement simple à utiliser par l'ingénieur logiciel sur MPPA® a été proposée, implémentée et validée. La parallélisation dans les nœuds multi-cœurs a été ensuite proposée par un environnement multitâche très efficace du fait qu'il utilise des mécanismes de synchronisation transactionnelle et sans exclusions mutuelles. Ces deux contributions ont ensuite permis d'élaborer des systèmes plus complexes mais cependant elles devaient être de maturité suffisamment élevées pour être utilisées

dans les prochaines contributions. Basée sur un modèle de flux de données hiérarchique et statique, une génération de code automatique visant le processeur MPPA® est réalisée. Cette contribution met en avant l'exploitation d'un modèle de calcul hiérarchique pour viser une architecture implémentant plusieurs niveaux hiérarchiques de parallélisme. Un environnement d'exécution de modèle de flux de données dynamique, proposé par l'équipe VAADER dans le laboratoire IETR, a été porté et adapté sur le MPPA®. Cet environnement permet de placer et ordonnancer des applications de flux de données paramétriques à l'exécution sur MPPA® de manière autonome. Un nouvel environnement distribué pour permettre l'exécution d'applications OpenVX sur un processeur massivement parallèle a été proposé. OpenVX est une interface de programmation standard et moderne qui permet de déployer du calcul sur un ou plusieurs accélérateurs depuis un hôte pour les applications de vision par ordinateur et de réseaux de neurones en inférence. L'environnement a été construit en partant de rien, uniquement de la spécification des fonctions du consortium Khronos et vise une exécution du graphe OpenVX très basse latence (implémentation hautement concurrentielle et asynchrone). L'environnement effectue l'optimisation de l'application automatiquement à l'exécution en appliquant des techniques telles que le pré chargement de données pour masquer les latences d'accès aux mémoires externes, ainsi que la fusion de nœuds de calcul pour éviter les copies de données en mémoire externe et éviter les problèmes de bande passante mémoire. En effet, sur les architectures massivement parallèles, le bouchon de performance est souvent situé sur la bande passante de la mémoire externe au processeur. C'est pour cela que des méthodes de fusion de calcul avancées sont appliquées afin d'augmenter la localité temporelle et spatiale de l'application. Diverses implémentations d'applications embarquées et hautes performances ont été aussi proposées et automatiquement optimisées sur le processeur MPPA® d'un point de vue utilisateur. Ces applications, telles que le stencil 3D et la transformée de Fourier distribuée, sont très concurrentielles et difficiles à mettre en œuvre.

Pour conclure, toutes les contributions qui sont présentées dans cette thèse ont été réalisées sur la véritable machine MPPA® seconde génération, testées et validées. De tels travaux ont impliqué des séances de déboguages intensives pour comprendre pourquoi la contribution ne fonctionnait pas afin de faire fonctionner la proposition finale. C'est pourquoi, tous les systèmes, applications, et librairies optimisées, présentés dans ces travaux de thèse sur MPPA® ont été limités par ma capacité à les observer.

List of Figures

List of Tables

Glossary

ABC Architecture Benchmark Computer. 46

ABI Application Binary Interface. 179

ADAS Advanced Driver-Assistance System. 3, 14

AI Arithmetic Intensity. 172, 184

AMD Advanced Micro Devices. 144, 145

AOS Asynchronous One-Sided. 70–72, 77, 79, 82–85, 90–93, 107, 108, 131, 133, 135, 156, 163, 167, 171, 177, 180, 181, 183

API Application Programming Interface. 5, 27, 29, 32–34, 51–53, 55, 65, 69–72, 77, 81, 92, 96, 98, 103, 106–108, 130, 131, 133, 135, 141, 142, 145, 146, 148, 163, 167, 170, 180, 187, 189, 193

ARM Advanced Reduced Instruction Set Computer (RISC) Machine. 14

ARMCI Aggregate Remote Memory Copy Interface. 53

BDF Boolean DataFlow. 40

BF Best-Fit. 44, 46

BLAS Basic Linear Algebra Subprograms. 93

BLODI BLOck DIagram compiler. 35

BPDF Boolean Parametric DataFlow. 40

BSP Bulk Synchronous Parallel. 165

CC Compute Cluster. 8, 17–19, 24, 25, 33, 49, 51, 61, 63, 69–71, 76–79, 82–84, 86, 88–91, 95, 96, 99, 101, 106, 110, 111, 113, 114, 116, 118–121, 124–127, 130, 131, 134–139, 142, 144–149, 151–153, 155–159, 161–165, 168, 170–174, 176–180, 183–185, 188, 189, 191, 192

CEA Centre des Energies Atomiques. 17

CMA Continuous Memory Allocator. 91

CNN Convolutional Neural Network. 7, 8, 54, 77, 88, 93, 165, 181–185, 189

CPU Central Processing Unit. 6, 7, 12–14, 16, 17, 19, 21, 22, 24, 32, 45, 54–57, 62, 77, 95, 96, 99, 103, 105, 107, 116, 119, 120, 125, 129, 138, 139, 142, 144, 145, 147, 148, 164, 165, 167, 184, 188, 190

CSDF Cyclo-Static Dataflow. 37, 38, 49

CTA Compositional Temporal Analysis. 115

CUDA Compute Unified Device Architecture. 33, 34

DAG Directed Acyclic Graph. 37–39, 43, 45, 46, 120, 134, 145, 149, 189

DCB Data Center Bridging. 52, 58

DDR Double Data Rate. 15, 18, 20, 33, 54, 71, 83, 84, 86, 88, 120, 124, 131, 135, 145, 166, 171–173, 176, 177, 183–185

DFS Depth-First Search. 42, 149

DFT Discrete Fourier Transform. 174

DMA Direct Memory Access. 5, 14, 18–20, 24, 25, 34, 54, 57–60, 69, 70, 72, 78–89, 91, 92, 95, 98, 103, 105, 107, 108, 110, 120, 130, 131, 135, 139, 141, 154, 155, 166, 167, 181, 183, 185, 187, 190–192, 202

DPN Dataflow Process Network. 35–37, 48

DSL Domain Specific Language. 45, 141, 145

DSM Distributed Shared Memory. 24, 25, 28, 33, 82, 91, 92, 155, 166

DSP Digital Signal Processor. 4, 6, 12–14, 48, 54, 118, 120, 124, 125, 129

DSSF Deterministic SDF with Shared FIFOs. 39

ELF Executable and Linkable Format. 22, 25, 45, 108

Eot End-of-Transfer. 20, 79, 80, 82, 84, 85, 183

FF First-Fit. 44, 46

FFT Fast Fourier Transform. 88, 165, 174–176, 178–181, 184, 185, 189

FIFO First-In-First-Out queue. 18, 35–37, 39, 47, 48, 60, 62, 71, 80, 83, 86, 90, 115, 133, 135

FLOPS Floating Point Operations per Second. 13, 17

FPGA Field-Programmable Gate Array. 12, 54, 144, 145

fps Frames per second. 125, 126, 137–139, 160, 161, 163

FPU Floating Point Unit. 18, 20

FSM Finite-State Machine. 35

GCC GNU Compiler Collection. 22, 29, 30, 34, 44, 61, 63, 95, 99, 106, 107, 112, 114, 124, 145, 179, 188, 191, 200

GCC GNU Compiler Collection. 106, 108

GDDR Graphical Double Data Rate. 15

GEMM General Matrix Multiply. 114

GOT Global Offset Table. 101

GPS Global Positioning System. 3

GPU Graphics Processing Unit. 12–14, 32–34, 52, 54, 107, 142, 144, 145, 148, 167, 171, 184, 185, 192

GRT Global RunTime. 47, 48, 131, 132, 134, 135, 138, 139

HAL Hardware Abstraction Layer. 11, 24

HBM High Bandwidth Memory. 15, 173

HBW Halo Bandwidth. 171

HPC High-Performance Computing. 51–54, 58, 70, 141, 158

IBSDF Interface-Based SDF. 38–40, 45, 49, 115–117, 120, 121, 123, 126, 128, 129, 141

IBTA Infiniband Trade Association. 52

IETR Institute of Electronics and Telecommunications of Rennes. 5–7, 45, 188, 198, 201, 205

ILP Instruction-Level Parallelism. 13, 25, 118, 185

IO Input/Output Subsystem. 17–20, 25, 49, 63, 71, 77, 83, 84, 90, 91, 95, 96, 98, 99, 106, 111, 124, 130–132, 138, 139, 142, 144, 145, 147, 150, 164, 188, 189, 192

IOCTL Input/Output Control. 21, 91

IOPS Input/Output Operation per Second. 75, 90, 93, 134, 147, 188

IoT Internet of Things. 3

IP Intellectual Property. 144

IPC Inter-Process Communication. 69, 71, 144

IR Intermediate Representation. 149

ISA Instruction Set Architecture. 60, 63

KANN Kalray Neural Network. 82, 114, 165, 182

KPN Kahn Process Network. 35, 49

LBM Lattice Boltzmann Method. 166–168, 171–173, 184

LLC Last Level Cache. 59

LLVM Low-Level Virtual Machine. 191

LRT Local RunTime. 47, 48, 130–137

LRU Least Recently Used. 61

MCU Micro-controller Unit. 15

MIMD Multiple Instructions, Multiple Data. 13

MKL Math Kernel Library. 5

MLUPS Mega Lattice Updates per Second. 171–173

MMU Memory Management Unit. 18, 20–25, 28, 52, 57, 59, 62, 91, 98, 104, 155

MoC Model of Computation. 35

MPI Message Passing Interface. 28, 52, 53, 56, 65, 70, 92, 120, 154

MPPA Multi-Purpose Processor Array. 5–8, 11, 12, 17–20, 23–25, 27, 31, 33, 48, 49, 51, 54, 60, 61, 65, 69–71, 79, 82, 83, 86, 90–93, 95, 96, 99, 101, 106, 107, 110, 116, 119, 120, 124–131, 135, 137–139, 141–146, 148, 156, 161, 163–165, 170–174, 176–178, 181–185, 187–192, 196, 197, 199, 200, 202, 204, 205

MPSoC Multiprocessor System-on-Chip. 4, 5, 8, 12, 13, 27, 33, 37, 45, 46, 48, 61, 71, 124, 128, 129, 145, 164, 182, 184

NMTR New Multi-Threading Runtime. 95–98, 101, 103–106, 108, 110, 112, 114

NoC Network on Chip. 8, 16, 18–20, 24, 25, 49, 65, 69–72, 77–92, 108, 126, 127, 132–134, 138, 145, 146, 148, 158, 176, 177, 180, 181, 183, 185, 187, 190

NORMA No Remote Memory Access. 16

NUMA Non-Uniform Memory Access. 13, 16, 21, 28, 57

OFA Open-Fabrics Association. 52

OpenMP Open Multi-Processing. 116, 129

Orcc Open RVC-CAL Compiler. 129

OS Operating System. 11, 12, 20–25, 28–31, 42, 71, 79, 84, 90, 91, 95, 97, 98, 124, 145

PCIE Peripheral Component Interconnect Express. 21, 49, 69, 71, 98, 124, 143, 150, 183

PDF Particle Distribution Function. 166

PE Processing Element. 12, 19, 20, 24, 25, 31, 33, 42, 43, 46, 47, 49, 54, 69–72, 75, 77–82, 84, 85, 90, 98–101, 103–106, 108, 110, 111, 113–119, 126–128, 130–139, 146, 148, 155–158, 161, 162, 170, 177–180

PGAN Pairwise Grouping of Adjacent Nodes. 121

PGAS Partitioned-Global-Address-Space. 53, 65

PIC Position Independent Code. 101, 147

PiMM Parameterized and Interfaced dataflow Meta-Model. 41, 49, 115, 129, 141, 198

PiSDF Parameterized and Interfaced SDF. 40, 41, 45–47, 115, 129–131, 134, 137–139

PREESM Parallel and Real-time Embedded Executives Scheduling Method. 6, 27, 45, 46, 49, 115–118, 120, 130, 131, 182, 188, 192, 201

PSDF Parameterized SDF. 40

PSO Partial Store Order. 60

QoS Quality-of-Service. 20, 70

QPI QuickPath Interconnect. 16, 58

RAM Random Access Memory. 14, 15

RAW Read-After-Write. 60, 74, 103, 112, 127, 135, 142, 147, 164, 178, 192

RDDP Remote Direct Data Placement. 52

RDMA Remote Direct Memory Access. 52, 57, 58, 60, 65, 69, 71–81, 83, 84, 86–88, 91, 92, 108, 110, 113, 114, 120, 124–127, 135, 139, 144–146, 148, 149, 151–154, 157, 158, 160–164, 168, 170, 176–178, 180, 183, 185, 189, 190, 192

RISC Reduced Instruction Set Computer. 4, 14

RM Resource Manager. 19, 79, 82, 84, 90, 99, 106

RMO Relaxed Memory Order. 60

RoCE RDMA over Converged Ethernet. 52, 65

RR Round Robin. 79, 105, 134, 135

RTOS Real-Time Operating System. 79, 191

RV Repetition Vector. 36–40, 46, 49, 117–119, 121, 123, 124, 128

SADF Scenario-Aware Dataflow. 40

SDF Synchronous Dataflow. 37–40, 42, 49, 115, 116, 120, 121, 123, 129, 141, 142

SDFG Synchronous Dataflow Graph. 35

SIMD Single Instruction, Multiple Data. 12, 13, 18, 25, 34, 113, 141, 144, 179, 185, 189

SIMT Single Instruction, Multiple Threads. 12, 13, 192

SISD Single Instruction, Single Data. 12

SMEM Shared Memory. 18, 19, 83, 86, 88, 89

SMP Symmetric Multi-Processor system. 8, 16, 21, 25, 27–29, 34, 49, 69, 70, 72, 73, 113, 118, 176

SoC System-on-Chip. 7, 20, 25, 33, 128, 145, 183, 184

SPDF Schedulable Parametric Dataflow. 40

SPIDER Synchronous Parameterized Interfaced Dataflow Embedded Runtime. 27, 35, 46–49, 114, 129–132, 134, 135, 137–139, 192, 201, 202

SPMD Single Program, Multiple Data. 52, 53

SR Single-Rate. 37, 45, 46, 142, 149

SRDAG Single-Rate DAG. 37–39, 43

sRIO Serial Rapid Input-Output. 58

TCM Tightly Coupled Memory. 15

TI Texas Instruments. 4, 6, 48, 124–126, 132, 154

TLB Translation Lookaside Buffer. 22, 25, 44, 59

TLS Thread Local Storage. 22, 99, 101, 107, 114

TSO Total Store Order. 60

TV Television. 3

UMA Uniform Memory Access. 16, 28, 31

UML Unified Modeling Language. 34

USB Universal Serial Bus. 21

VAADER Video Analysis and Architecture Design for Embedded Resources. 5–7, 188, 198, 201, 205

VIA Virtual Interface Architecture. 52

VLIW Very Long Instruction Word. 13, 17, 18, 24, 25, 51, 61–65, 99, 103, 106, 125, 130, 179, 180, 185, 196

VMM Virtual Machine Monitor. 20

WCET Worst-Case Execution Time. 15, 44, 48, 64

Personal Publications

[HDD18] Hascoët, Julien and de Dinechin, Benoît Dupont and Desnos, Karol and Nezan, Jean-François. A Distributed Framework for Low-Latency OpenVX over the RDMA NoC of a Clustered Manycore. 2018 IEEE High Performance extreme Computing Conference (HPEC), 2018 Conference.

[MHDM18] Miomandre, Hugo and Hascoët, Julien and Desnos, Karol and Martin, Kevin JM and de Dinechin Kalray, Benoît Dupont and Nezan, Jean-François. Embedded Runtime for Reconfigurable Dataflow Graphs on Manycore Architectures. Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms.

[HDG17] Hascoët, Julien and de Dinechin, Benoît Dupont and de Massas, Pierre Guironnet and Ho, Minh Quan. Asynchronous one-sided communications and synchronizations for a clustered manycore processor. Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia, 2017 Conference.

[HDN17] Hascoët, Julien and Desnos, Karol and Nezan, Jean-François and de Dinechin, Benoît Dupont. Hierarchical Dataflow Model for efficient programming of clustered manycore processors. Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference.

[MHD17] Miomandre, Hugo and Hascoet, Julien and Desnos, Karol and Martin, Kevin and de Dinechin, Benoit Dupont and Nezan, Jean Francois. Demonstrating the SPIDER Runtime for Reconfigurable Dataflow Graphs Execution onto a DMA-based Manycore Processor. IEEE International Workshop on Signal Processing Systems, 2017 Conference.

[HOT17] Ho, Minh-Quan and Obrecht, Christian and Tourancheau, Bernard and de Dinechin, Benoît Dupont and Hascoet, Julien. Improving 3D Lattice Boltzmann Method stencil with asynchronous transfers on many-core processors. 36th IEEE International Performance Computing and Communications Conference (IPCCC 2017).

[HNE15] Hascoet, Julien and Nezan, Jean-Francois and Ensor, Andrew and de Dinechin, Benoît Dupont. Implementation of a fast Fourier transform algorithm onto a manycore processor. Design and Architectures for Signal and Image Processing (DASIP), 2015 Conference.

Bibliography

[air18] *Airbus*, 2018. http://www.airbus.com. 3

[Aja09] Jasmin Ajanovic. Pci express 3.0 overview. In *Proceedings of Hot Chip: A Symposium on High Performance Chips*, 2009. 69, 143

[ALP97] Marleen Adé, Rudy Lauwereins, and JA Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *Proceedings of the 34th annual Design Automation Conference*, pages 64–69. ACM, 1997. 44

[BA14] F Brill and E Albuz. Nvidia visionworks toolkit. In *GPU Technology Conference*, 2014. 144

[BAMJ13] S. C. Brunet, C. Alberti, M. Mattavelli, and J. W. Janneck. Design space exploration of high level stream programs on parallel architectures: a focus on the buffer size minimization and optimization problem. In *Image and Signal Processing and Analysis (ISPA)*, pages 738–743. IEEE, 2013. 44

[Bar93] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM, 1993. 95

[BB01] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, 2001. 40

[BB07] Peter Brucker and P Brucker. *Scheduling algorithms*, volume 3. Springer, 2007. 42, 43, 127

[BČH09] Mathieu Bouchard, Mirjana Čangalović, and Alain Hertz. About equivalent interval colorings of weighted graphs. *Discrete Applied Mathematics*, 157(17):3615–3624, 2009. 44, 45

[BCL+95] Eric A Brewer, Frederic T Chong, Lok T Liu, Shamik D Sharma, and John D Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 42–53. ACM, 1995. 70, 80

[Ber07] Gérard Berry. SCADE: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007. 49

[Ber18] Berkeley. *Latency Numbers Every Programmer Should Know*, 2018. https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html. 14

[BFGL13] Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. Bpdf: A statically analyzable dataflow model with integer and boolean parameters. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–10. IEEE, 2013. 40

[BL93] Joseph Tobin Buck and Edward A Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432. IEEE, 1993. 40, 42, 44

[Blo17] Nvidia Developer Blog. *Inside Volta: The World's Most Advanced Data Center GPU*, 2017. https://devblogs.nvidia.com/inside-volta/. 13

[BML99] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(2):151–166, 1999. 142, 149

[BML12] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. *Software synthesis from dataflow graphs*, volume 360. Springer Science & Business Media, 2012. 121, 123

[Bon08] Dan Bonachea. Gasnet specification. Technical report, 2008. 53

[BPG01] Darius Buntinas, Dhabaleswar K Panda, and William Gropp. Nic-based atomic remote memory operations in myrinet/gm. In *IN MYRINET/GM," IN WORKSHOP ON NOVE USES OF SYSTEM AREA NETWORKS (SAN1)*. Citeseer, 2001. 58

[BYY+16] Janki Bhimani, Jingpei Yang, Zhengyu Yang, Ningfang Mi, Qiumin Xu, Manu Awasthi, Rajinikanth Pandurangan, and Vijay Balakrishnan. Understanding performance of i/o intensive containerized applications for nvme ssds. In *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*, pages 1–8. IEEE, 2016. 93

[CAR14] Ivano Cerrato, Mauro Annarumma, and Fulvio Risso. Supporting fine-grained network functions through intel dpdk. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 1–6. IEEE, 2014. 97

[CCJM97] Nianzheng Cao, Shiyi Chen, Shi Jin, and Daniel Martinez. Physical symmetry and lattice symmetry in the lattice Boltzmann method. *Physical Review E*, 55(1):R21, 1997. 166

[CCP+10] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010. 28, 53

[CCS+08] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. Maps: an integrated framework for mpsoc application parallelization. In *Proceedings of the 45th annual Design Automation Conference*, pages 754–759. ACM, 2008. 49

[CDG+14] Loïc Cudennec, Paul Dubrulle, François Galea, Thierry Goubier, and Renaud Sirdey. Generating code and memory buffers to reorganize data on many-core architectures. In *Procedia Computer Science*, volume 29, pages 1123–1133, 2014. 151

[CEL+03] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (rsvptm). In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 141. IEEE Computer Society, 2003. 76

[CH89] JE Cooling and TS Hughes. The emergence of rapid prototyping as a real-time software development tool. In *Software Engineering for Real Time Systems, 1989., Second International Conference on*, pages 60–64. IET, 1989. 45

[Che00] Yao-Ting Cheng. Autoscaling radix-4 fft for tms320c6000. *application report SPRA654*, 2000. 174, 175

[CJVDP08] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008. 29

[CLA13] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, 2013. 49

[CPC16] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016. 184

[CPG+] M. Chavarras, F. Pescador, M. J. Garrido, E. Juairez, and C. Sanz. A multicore DSP HEVC decoder using an actorbased dataflow model and OpenMP. 61(2):236–244. 49, 120

[CRP+16] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini. Pulp: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision. *Journal of Signal Processing Systems*, 84(3):339–354, 2016. 6

[CSWZ16] Jiecao Chen, He Sun, David Woodruff, and Qin Zhang. Communication-optimal distributed clustering. In *Advances in Neural Information Processing Systems*, pages 3727–3735, 2016. 124

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. 174

[CTK+09] David Cohen, Thomas Talpey, Arkady Kanevsky, Uri Cummings, Michael Krause, Renato Recio, Diego Crupnicoff, Lloyd Dickman, and Paul Grun. Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options. In *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, pages 123–130. IEEE, 2009. 52

[dDAB+13] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013. 49, 151

[dDdML+13] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013. 31, 69, 70

[DDNMK17a] Hamza Deroui, Karol Desnos, Jean-Francois Nezan, and Alix Munier-Kordon. Relaxed subgraph execution model for the throughput evaluation of ibsdf graphs. In *International Conference on Embedded Computer Systems: Architecture, Modeling and Simulation SAMOS*, 2017. 115

[DDNMK17b] Hamza Deroui, Karol Desnos, Jean-Francois Nezan, and Alix Munier-Kordon. Throughput evaluation of dsp applications based on hierarchical dataflow models. In *Proceedings of the 50th International Symposium on Circuits and Systems. ISCAS*, 2017. 115

[DDRV17] Benoit Dupont De Dinechin, Marta RYBCZYNSKA, and RAY Vincent. Atomic instruction having a local scope limited to an intermediate cache level, September 7 2017. US Patent App. 15/452,073. 63

[Des14] Karol Desnos. *Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs*. PhD thesis, INSA de Rennes, 2014. 40, 41, 44, 45, 46, 122, 207

[DGCDM97] Eddy De Greef, Francky Catthoor, and Hugo De Man. Array placement for storage size reduction in embedded multimedia systems. In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, pages 66–75. IEEE, 1997. 45

[DLP03] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003. 13

[dM09] P Guironnet de Massas. *Etude de méthodes et mécanismes pour un acces transparent et efficace aux données dans un systeme multiprocesseur sur puce*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2009. 24

[Doc07] GCC Documentation. *Built-in functions for atomic memory access*, 2007. https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html. 61, 65

[DPN+13] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 41–48. IEEE, 2013. 41, 115, 129

[DPNA15] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. Buffer merging technique for minimizing memory footprints of synchronous dataflow specifications. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1111–1115. IEEE, 2015. 44

[DPNA16] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. Distributed memory allocation technique for synchronous dataflow graphs. In *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*, pages 45–50. IEEE, 2016. 46, 119, 124

[Dre03] Ulrich Drepper. Elf handling for thread-local storage. Technical report, Technical report, Red Hat, Inc., 2003. URL http://people. redhat. com/-drepper/tls. pdf. 6.4. 1, 2003. 22

[DS95] Michael Dolle and Manfred Schlett. A cost-effective risc/dsp microprocessor for embedded systems. *IEEE Micro*, 15(5):32–40, 1995. 4

[Exe02] UML Executable. A foundation for model-driven architecture. *Stephen J*, 2002. 35

[Fei95] Karl Feind. Shared memory access (shmem) routines. *Cray Research*, 1995. 53

[FGP12] Pascal Fradet, Alain Girault, and Peter Poplavko. Spdf: A schedulable parametric data-flow moc. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 769–774. EDA Consortium, 2012. 40

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972. 12

[G+11] Khronos OpenCL Working Group et al. The opencl specification version 1.1. *www.khronos.org/registry/cl/specs/opencl-1.1.pdf*, 2011. 32, 144

[G+17] Khronos Vision Working Group et al. The openvx specification v1. 1. *Web: www.khronos.org/registry/OpenVX/specs/1.1/OpenVX_ Specification_ 1_ 1.pdf*, 2017. 7, 141, 142, 143, 149, 154, 155, 203, 209

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995. 132

[Gid17] Radhakrishna Giduthuri. *The OpenVX Safety Critical*, 2017. https://www.khronos.org/registry/OpenVX/specs/1.1_SC/OpenVX_Specification_SC_1_1.pdf. 147

[Gil74] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974. 35

[GL04] Alexandros V Gerbessiotis and Seung-Yeop Lee. Remote memory access: A case for portable, efficient and library independent parallel programming. *Scientific Programming*, 12(3):169–183, 2004. 53

[GMRdD18] Amaury Graillat, Matthieu Moy, Pascal Raymond, and Benoît Dupont de Dinechin. Parallel code generation of synchronous programs for a many-core architecture. In *Design, Automation and Test in Europe*, 2018. 49

[GNP90] David Gelernter, Alexandru Nicolau, and David A Padua. *Languages and compilers for parallel computing*. Pitman, 1990. 53

[Gor04] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, January 2004. 80

[GP16] Radhakrishna Giduthuri and Kari Pulli. Openvx: a framework for accelerating computer vision. In *SIGGRAPH ASIA 2016 Courses*, page 14. ACM, 2016. 144

[Gro18] Khronos Group. *Khronos Website*, 2018. 34

[HDB+12] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. Leveraging mpi's one-sided communication interface for shared-memory programming. *Recent advances in the message passing interface*, pages 132–141, 2012. 28, 120

[HdDdMH17] Julien Hascoët, Benoît Dupont de Dinechin, Pierre Guironnet de Massas, and Minh Quan Ho. Asynchronous one-sided communications and synchronizations for a clustered manycore processor. In *Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia*, pages 51–60. ACM, 2017. 93, 148, 153

[HDN+12] Julien Heulot, Karol Desnos, J-F Nezan, Maxime Pelcat, Mickaël Raulet, Hervé Yviquel, P-L Lagalaye, and J-C Le Lann. An experimental toolchain based on high-level dataflow models of computation for heterogeneous mpsoc. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pages 1–2. IEEE, 2012. 46

[HDT+15] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote memory access programming in mpi-3. *ACM Transactions on Parallel Computing*, 2(2):9, 2015. 72, 120

[HJB84] Michael Heideman, Don Johnson, and C Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984. 174

[HNEdD15] Julien Hascoet, Jean-Francois Nezan, Andrew Ensor, and Benoît Dupont de Dinechin. Implementation of a fast fourier transform algorithm onto a manycore processor. In *Design and Architectures for Signal and Image Processing (DASIP), 2015 Conference on*, pages 1–7. IEEE, 2015. 174, 176

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 59

[HPD+14] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-Francois Nezan, and Slaheddine Aridhi. Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 167–171. IEEE, 2014. 7, 21, 40, 46, 48, 129, 132, 188

[IHIY14] Khaled Z Ibrahim, Paul H Hargrove, Costin Iancu, and Katherine Yelick. An evaluation of one-sided and two-sided communication paradigms on relaxed-ordering interconnect. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1115–1125. IEEE, 2014. 56, 57, 70, 74

[Ins17] Texas Instrument. *Multicore DSP-ARM Keystone II System-on-Chip (SoC)*, 2017. http://www.ti.com/lit/ds/symlink/66ak2h12.pdf. 14

[Int18] Intel. *Intel QuickPath Interconnect*, 2018. https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html. 16

[iW18] National Supercomputing Center in Wuki. *Sunway TaihuLight Supercomputer System*, 2018. http://www.nsccwx.cn/wxcyw/soft1.php?word=soft&i=46. 13

[JH16] Benoit Dupont de Dinechin Julien Hascoet. Kalray MPPA Asynchronous One-Sided Library, 2016. 167

[Joh73] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973. 44

[Kal] Kalray. Deep Learning for High-Performance Embedded Applications. 182

[KCDZ94] Peter J Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter*, volume 1994, pages 23–36, 1994. 16, 18, 24, 25, 28, 57

[KEHS+15] Hee-Seok Kim, Izzat El Hajj, John Stratton, Steven Lumetta, and Wen-Mei Hwu. Locality-centric thread scheduling for bulk-synchronous programming models on cpu architectures. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 257–268. IEEE, 2015. 165

[KJLV61] John L Kelly Jr, Carol Lochbaum, and Victor A Vyssotsky. A block diagram compiler. *Bell System Technical Journal*, 40(3):669–676, 1961. 35

[KL95] David J King and John Launchbury. Structuring depth-first search algorithms in haskell. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 344–354. ACM, 1995. 43, 149

[KP03] J Nieplocha V Tipparaju M Krishnan and G Santhanaraman DK Panda. Optimizing mechanisms for latency tolerance in remote memory access communication on clusters. In *IEEE International Conference on Cluster Computing*, page 138. IEEE, 2003. 57

[KS10] Fredrik Berg Kjolstad and Marc Snir. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, page 4. ACM, 2010. 144

[KVL91] Bo Kågström and Charles F Van Loan. *GEMM-based level-3 BLAS*. Cornell Theory Center, Cornell University, 1991. 114

[Kwo97] Yu-Kwong Kwok. *High-performance algorithms for compile-time scheduling of parallel processors*. PhD thesis, 1997. 42, 46, 134

[LG96] Doug Lea and Wolfram Gloger. A memory allocator, 1996. 44

[LH89] Edward A Lee and Soonhoi Ha. Scheduling strategies for multiprocessor real-time dsp. In *Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'(GLOBECOM), 1989. IEEE*, pages 1279–1283. IEEE, 1989. 42

[LJW+04] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabaleswar K Panda, David Ashton, Darius Buntinas, William Gropp, and Brian Toonen. Design and implementation of mpich2 over infiniband with rdma support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 16. IEEE, 2004. 56, 60, 81

[LM87] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. 37, 38, 39, 134, 142

[LMS04] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. In *International Symposium on Distributed Computing*, pages 117–131. Springer, 2004. 63

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(3):257–279, 1999. 15

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2), 2008. 13

[LP95] Edward A Lee and Thomas M Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995. 35

[LPF13] Thierry Lepley, Pierre Paulin, and Eric Flamand. A novel compilation approach for image processing graphs on a many-core platform with explicitly managed memory. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 6. IEEE Press, 2013. 55

[MB00] Praveen K Murthy and Shuvra S Bhattacharyya. Shared memory implementations of synchronous dataflow specifications. In *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pages 404–410. IEEE, 2000. 45

[MCASJ09] John Mellor-Crummey, Laksono Adhianto, William N. Scherer, III, and Guohua Jin. A New Vision for Coarray Fortran. In *Proc. of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5:1–5:9, 2009. 53

[MGL+16] Markus Maurer, J Christian Gerdes, Barbara Lenz, Hermann Winner, et al. *Autonomous driving*. Springer, 2016. 3

[MHH+85] Kichie Matsuzaki, Seiji Hata, Junichi Hamano, Youichi Kurashima, and Masahiro Torii. Petri-net structured sequence control language with grafcet-like graphical expression for programmable controllers. In *Proc. IECON'85*, pages 433–438, 1985. 35

[MHTR08] Keijo Mattila, Jari Hyväluoma, Jussi Timonen, and Tuomo Rossi. Comparison of implementations of the lattice-Boltzmann method. *Computers & Mathematics with Applications*, 55(7):1514–1524, 2008. 170

[Mic02] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002. 63

[MNW14] James Mistry, Matthew Naylor, and Jim Woodcock. Adapting freertos for multicores: An experience report. *Software: Practice and Experience*, 44(9):1129–1154, 2014. 21

[MP92] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *ACM SIGOPS Operating Systems Review*, 26(2):108, 1992. 95, 113

[MRPC10] Miguel Masmano, Ismael Ripoll, S Peiró, and A Crespo. Xtratum for leon3: an open source hypervisor for high integrity systems. In *European Conference on Embedded Real Time Software and Systems. ERTS2*, volume 2010, 2010. 24

[MRSD16] Kevin JM Martin, Mostafa Rizk, Martha Johanna Sepulveda, and Jean-Philippe Diguet. Notifying memories: a case-study on data-flow applications with noc interfaces implementation. In *Proceedings of the 53rd Annual Design Automation Conference*, page 35. ACM, 2016. 58, 77

[MSBCP14] Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price. On the performance portability of structured grid codes on many-core computer architectures. In *Supercomputing*, pages 53–75. Springer, 2014. 173

[NC99] Jarek Nieplocha and Bryan Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Parallel and Distributed Processing*, pages 533–546, 1999. 53

[new] newlib. https://github.com/bminor/newlib. 96, 98

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. 53

[NTKP06] Jarek Nieplocha, Vinod Tipparaju, Manojkumar Krishnan, and Dhabaleswar K Panda. High performance remote memory access communication: The armci approach. *International Journal of High Performance Computing Applications*, 20(2):233–253, 2006. 57, 69

[Nvi15] Nvidia. *Nvidia Tegra X1*, 2015. http://www.nvidia.com/object/tegra-x1-processor.html. 14

[Nvi17] Nvidia. *Nvidia Drive PX 2*, 2017. https://www.nvidia.com/en-us/self-driving-cars/drive-px/. 14

[Nvi18] Nvidia. *GPU-Accelerated Libraries for Computing*, 2018. https://developer.nvidia.com/gpu-accelerated-libraries. 34

[O+89] John K Ousterhout et al. *Tcl: An embeddable command language*. Citeseer, 1989. 81

[Oga95] Takeshi Ogasawara. An algorithm with constant execution time for dynamic storage allocation. In *Real-Time Computing Systems and Applications, 1995. Proceedings., Second International Workshop on*, pages 21–25. IEEE, 1995. 44

[Olo16] Andreas Olofsson. Epiphany-v: A 1024 processor 64-bit RISC system-on-chip. *CoRR*, abs/1610.01832, 2016. 6, 51

[Ope13] ARB OpenMP. Openmp 4.0 specification, june 2013, 2013. 34

[Ost95] J.S. Ostroff. Abstraction and composition of discrete real-time systems. *Proc. of CASE*, 95:370–380, 1995. 115

[OTK15] Christian Obrecht, Bernard Tourancheau, and Frédéric Kuznik. Performance Evaluation of an OpenCL Implementation of the Lattice Boltzmann Method on the Intel Xeon Phi. *Parallel Processing Letters*, 25(03):1541001, 2015. 167

[Pap16] Jean-Charles Papin. *A Scheduling and Partitioning Model for Stencil-based Applications on Many-Core Devices*. PhD thesis, Université Paris-Saclay, 2016. 42

[PAPN12] Maxime Pelcat, Slaheddine Aridhi, Jonathan Piat, and Jean-François Nezan. *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*. Springer, 2012. 115

[PBR09] Jonathan Piat, Shuvra S Bhattacharyya, and Mickaël Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 145–150. IEEE, 2009. 39, 41, 115, 116

[PJ09] Katalin Popovici and Ahmed Jerraya. Hardware abstraction layer. In *Hardware-dependent Software*, pages 67–94. Springer, 2009. 11, 24

[PMAN09] Maxime Pelcat, Pierrick Menuet, Slaheddine Aridhi, and Jean-François Nezan. Scalable compile-time scheduler for multi-core architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1552–1555. European Design and Automation Association, 2009. 46

[PNA10] M. Pelcat, J. F. Nezan, and S. Aridhi. Adaptive multicore scheduling for the LTE uplink. In *NASA/ESA Conference on Adaptive Hardware and Systems*, pages 36–43, 2010. 42

[Ras87] Richard F Rashid. Designs for parallel architectures. *Unix Review*, 5(4):36–43, 1987. 16

[RoC15] Mellanox, roce vs. iwarp competitive analysis, white paper, August 2015. 52

[Rus78] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978. 12, 13

[RVD+14] Erik Rainey, Jesse Villarreal, Goksel Dedeoglu, Kari Pulli, Thierry Lepley, and Frank Brill. Addressing system-level optimization with openvx graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 644–649, 2014. 144

[SAS15] Khushboo Singh, Mahfooz Alam, and Sushil Kumar Sharma. A survey of static scheduling algorithm for distributed computing system. *International Journal of Computer Applications*, 129(2), 2015. 43

[SEU+15] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 220–229, 2015. 17, 51

[SG12] Bogdan Spinean and Georgi Gaydadjiev. Implementation study of fft on multi-lane vector processors. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 815–822. IEEE, 2012. 175

[SGB06] Sander Stuijk, Marc Geilen, and Twan Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 899–904. IEEE, 2006. 44

[Sha03] Tom Shanley. *Infiniband Network Architecture*. Addison-Wesley Professional, 2003. 69, 74

[SHW11] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011. 22, 58, 59, 60

[SJA+13] Eric Stotzer, Ajay Jayaraj, Murtaza Ali, Arnon Friedmann, Gaurav Mitra, Alistair P Rendell, and Ian Lintault. Openmp on the low-power ti keystone ii arm/dsp system-on-chip. In *International Workshop on OpenMP*, pages 114–127. Springer, 2013. 124

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. 183

[SLwRMSD18] Andrew Oram Sandra Loosemore with Richard M. Stallman, Roland McGrath and Ulrich Drepper. *The GNU C Library Reference Manual*, 2018. https://www.gnu.org/software/libc/manual/pdf/libc.pdf. 21, 29, 44, 61

[SSKH13] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013. 47

[Suc01] Sauro Succi. *The lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press, 2001. 166

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005. 23

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972. 43

[TBG+13] Stavros Tripakis, Dai Bui, Marc Geilen, Bert Rodiers, and Edward A Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(3):83, 2013. 38, 39

[TGB+06] Bart D Theelen, Marc CW Geilen, Twan Basten, Jeroen PM Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 185–194. IEEE, 2006. 40

[THB14] Giuseppe Tagliavini, Germain Haugou, and Luca Benini. Optimizing memory bandwidth in openvx graph execution on embedded many-core accelerators. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pages 1–8. IEEE, 2014. 144

[THMB15] Giuseppe Tagliavini, Germain Haugou, Andrea Marongiu, and Luca Benini. Adrenaline: an openvx environment to optimize embedded vision applications on many-core accelerators. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, pages 289–296. IEEE, 2015. 144

[TIC13] *Texas Instruments: Tms320c6678*, 2013. ".http://www.ti.com/product/tms320c6678. 46

[Tor97] Linus Torvalds. Linux: a portable operating system. *Master's thesis, University of Helsinki, dept. of Computing Science*, 1997. 21

[TRBD01] Linus Torvalds and David Read By-Diamond. *Just for fun: The story of an accidental revolutionary*. Harper Audio, 2001. 21

[VCHP07] Karthikeyan Vaidyanathan, Lei Chai, Wei Huang, and Dhabaleswar K Panda. Efficient asynchronous memory copy operations on multi-core systems and i/oat. In *Cluster Computing, 2007 IEEE International Conference on*, pages 159–168. IEEE, 2007. 57

[VEMR14] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P Rendell. Programming the adapteva epiphany 64-core network-on-chip coprocessor. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 984–992. IEEE, 2014. 51

[VWM04] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109. ACM, 2004. 59

[VZT+18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv 1802.04730*, 2018. 144

[VZVDG15] Field G Van Zee and Robert A Van De Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):14, 2015. 93, 164

[Win56] Omar Wing. Ladder network analysis by signal-flow graph-application to analog computer programming. *IRE Transactions on Circuit Theory*, 3(4):289–294, 1956. 35

[WJNB95] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, pages 1–116. Springer, 1995. 22

[WM95] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995. 21

[WM14] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202. IEEE Press, 2014. 148

[Wol02] Wayne Wolf. What is embedded computing? *Computer*, 35(1):136–137, 2002. 3

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012. 34, 145

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. 4, 34, 127, 135, 141, 144, 153, 154, 158, 161, 164, 172

[ZK06] Christian Zinner and Wilfried Kubinger. Ros-dma: a dma double buffering method for embedded image processing with resource optimized slicing. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 361–372. IEEE, 2006. 107

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Contributions aux environnements d'exécution pour processeurs massivement parallèles appliqués aux applications embarquées et hautes performances

Nom Prénom de l'auteur : HASCOËT JULIEN

Membres du jury :
- Monsieur DUPONT DE DINECHIN Benoît
- Monsieur GIRAULT Alain
- Monsieur NEZAN Jean-François
- Monsieur DESNOS Karol
- Madame MUNIER-KORDON Alix
- Monsieur IRIGOIN François

Président du jury : Alix Munier Kordon

Date de la soutenance : 14 Décembre 2018

Reproduction de la these soutenue

☐ Thèse pouvant être reproduite en l'état
☒ Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 14 Décembre 2018

Signature du président de jury

Le Directeur,

M'hamed DRISSI

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

INSA RENNES

Title : Contributions to Software Runtime for Clustered Manycores Applied to Embedded and High-Performance Applications

Keywords: High-Performance Computing, Parallelisms, Communications, Dataflow, Embedded systems

The growing need for computing is more and more challenging, especially in the embedded system world with autonomous cars, drones, and smartphones. New highly parallel and heterogeneous processors emerge to answer this challenge. They operate in constrained environments with real-time requirements, reduced power consumption, and safety. Programming these new chips is a time-consuming and challenging task leading to huge software development costs. The Kalray MPPA® processor is a competitive example for low-power super-computing on a single chip. It integrates up to 288 VLIW cores grouped in 18 clusters, each fitted with shared local memory. These clusters are interconnected with a high-bandwidth network-on-chip, and DMA engines are used to communicate. This processor is used in this thesis for experimental results.

We propose the AOS library enabling high-performance communications and synchronizations of distributed local memories on clustered manycores. AOS provides 70% of the peak hardware throughput for transfers larger than 8 KB. We propose tools for the implementation of static and dynamic dataflow programs based on AOS to accelerate the parallel application developments onto clustered manycores. We propose an implementation of OpenVX for clustered manycores on top of AOS. OpenVX is a standard based on dataflow for the development of computer vision and neural network computing. The proposed OpenVX implementation includes automatic optimizations like data prefetch to overlap communications and computations, or kernel fusion to avoid the main memory bandwidth bottleneck. Results show super-linear speedups.

Titre : Contributions aux environnements d'exécution pour processeurs massivement parallèles et clustérisés appliqués aux applications embarquées et hautes performances

Mots clés : Calcul à hautes performances, Parallélismes, Communications, Flux-de-données, Systèmes embarqués

Le besoin en calculs est toujours plus important et difficile à satisfaire, spécialement dans le domaine de l'informatique embarquée qui inclue les voitures autonomes, drones et téléphones intelligents. Les systèmes embarqués doivent respecter des contraintes fortes de temps, de consommation et de sécurité. Les nouveaux processeurs parallèles et hétérogènes comme le MPPA® de Kalray utilisé dans cette thèse, doivent alors combiner haute performance et basse consommation. Pour cela, le MPPA® intègre 288 cœurs, regroupés en 18 clusters à mémoire locale partagée, un réseau sur puce et des moteurs DMA pour les communications. Ces processeurs sont difficiles à programmer, engendrant des coûts de développement importants. Cette thèse a pour objectif de simplifier leur programmation tout en optimisant les performances finales.

Nous proposons pour cela AOS, une librairie de communication et synchronisation haute performance gérant les mémoires locales distribuées des processeurs clustérisés. La librairie atteint 70% de la crête matérielle pour des transferts supérieurs à 8 KB. Nous proposons plusieurs outils de développement basés sur AOS et des modèles de programmation flux-de-données pour accélérer le développement d'applications parallèles pour processeurs clustérisés, notamment OpenVX qui est un nouveau standard pour les applications de vision et les réseaux de neurones. Nous automatisons l'optimisation de l'application OpenVX en faisant du pré-chargement de données et en les fusionnants, pour éviter le mur de la bande passante mémoire externe. Les résultats montrent des facteurs d'accélération super linéaires.