Transformations source-à-source pour l'optimisation de codes irréguliers et multithreads

Julien Jaeger

UNIVERSITÉ DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

Département de formation doctorale en informatique **École doctorale SVT**
UFR des Sciences

Source-to-source transformations for irregular and multithreaded code optimization

THÈSE

présentée et soutenue publiquement le 02/07/2012

pour l'obtention du

Doctorat
de l'université de Versailles / Saint-Quentin-en-Yvelines
(spécialité informatique)

par

Julien JAEGER

Composition du jury

Rapporteurs : Bernd Mohr, *Jülich Supercomputing Centre*
François Bodin, *Université de Rennes 1*

Examinateurs : Denis Barthou, *Université de Bordeaux 1*
Patrick Carribault, *CEA-DAM*
William Jalby, *Université de Versailles Saint-Quentin-En-Yvelines*
Erven Rohou, *INRIA Rennes*
Sid Touati, *Université de Nice Sophia Antipolis*

Laboratoire Parallélisme, Réseaux, Systèmes, Modélisation — UMR 8144

Laboratoire PRISM

Mis en page avec la classe thloria.

Table des matières

Table des figures

Chapitre 1

Introduction

Source-to-Source optimization is an efficient method to generate, from a basic implementation, a high performance program for the two main challenges that are irregular codes and heterogeneous implementation.

In the last decade, general purpose CPUs moved towards multi-core architectures, and the end of the increase in processors frequency marked a turning point obtaining the best performance of a single chip, achieved only when efficiently considering the parallelism inside the chip. The optimization process is now a paramount key to have continuously increasing speed-up on newest architectures. Parallelization on a single chip brings new problems to consider, with the integration of different cache level on the chip, and having several threads running simultaneously and accessing to shared resources. Such coexistence implies that the different levels of parallelism (vector, Instruction Level Parallelism, threads, memory access) interacts more than ever, and optimization for high performance should consider all levels. A second paradigm shift occurs with the generalization of hardware accelerators and heterogeneous machines, requiring expertise in all architectures composing the heterogeneous system when generating an efficient code for the target. The complication of hardware architectures provides many challenges in the HPC area, especially for irregular codes, whether irregular in data access or in control flow, since generating efficient version for such code on only one core remains difficult.

In this dissertation, we will provide methods to generate efficient codes from an initial implementation of irregular programs and heterogeneous parallelizations. The remaining of Chapter 1 presents the evolution of machine architecture from the first scalar computer to nowadays multi-core and heterogeneous systems. Then, since some code transformation is usually required to help the compiler to take advantage of the complex features of nowadays processors, the most used source-to-source optimizations and loop transformations are described. Finally, as the vectorization is one of the most difficult optimization to apply efficiently, we provide some insight into the hardware support for vector codes, dictating their own restrictions. Chapter 2 describes our CPC framework, extracting codelets from an irregular codes, optimizing these codelets regardless the overall program, then predicting the overall speed-up of the all system. We validate the framework on three real codes used in physics, genomic and molecular chemistry. In Chapter 3, we develop methods, with more or less complexity and memory impact, to address alignment issues, due to vectorization or bank conflicts. We apply our methods on symptomatic stencil cases, providing an algorithm to generate efficient codes for CPUs and GPUs. Parallelization techniques are discussed in Chapter 4 with the presentation of two works, one addressing the generation of parallelized codelets, the second scheduling sequential tasks on an heterogeneous system. To conclude, Chapter 5 will remind the contribution of the dissertation, and discuss the improvement and future development possible concerning the presented works.

1.1 Evolution of Processor Architectures

Evolutionary biologist Theodosius Dobzhansky defined *adaptation* as *"the evolutionary process whereby an organism becomes better able to live in its habitat or habitats"*. The evolution of computer architectures follows the same pattern : trying to be the fittest possible in the race to performance. Through history, computer architectures evolved with computer engineers trying to bypass the main issues encountered during this race. These main issues are qualified as "walls", and all major innovations were done to circumvent those three walls : the instruction wall, the memory wall and for the last decade, the thermal wall.

The instruction wall is the main focus of computer science since the first computer, and aims to execute a maximum of instructions in the minimum lapse of time and, to a certain extent, to do several instructions at the same time. In Section 1.1.1, evolution of the processing power is described to the first simple scalar processor to our vector processors, via pipelined, superscalar and VLIW processors. Increasing the processing power of processors, a gap widened between the speed of computations and memory transfers, hitting the second wall : the memory wall. Mechanisms to bypass this wall are evoked in Section 1.1.2, with the two different local memories that are the automatically managed caches and the manually managed scratchpad. For the last decade, the performance race hit a new wall, the thermal wall. Until then, processor performance followed the Moore's law (Fig. 1.1) stating that the number of transistors placed on an integrated circuit doubles every two years.

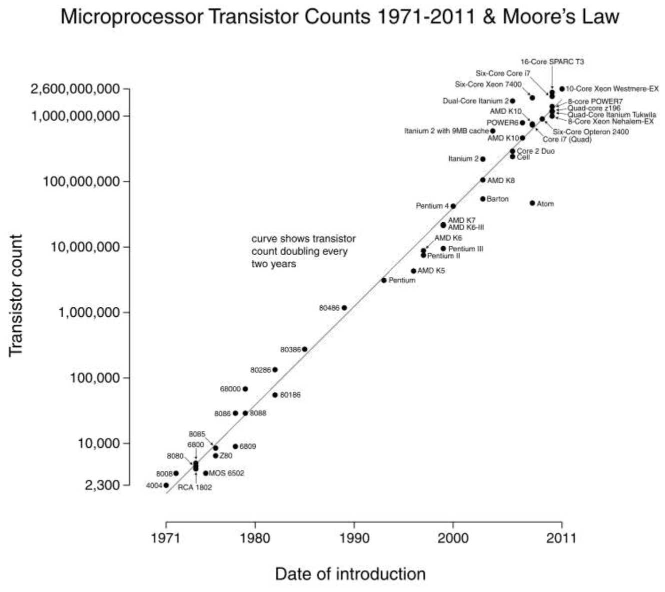


FIGURE 1.1 – Number of transistors on a single chip since 1971

However, this law did not foreseen that physical issue would be involved, and that heat dissipation would be inexorable. The thermal wall caused drastic changes in computer architectures. Coupled with the precedent encountered walls, paradigm shifts towards multi-core processors and heterogeneous systems took place as described in Section 1.1.3. Nowadays, several tradeoffs between these solutions are proposed in new computers, and one has to understand why and how they are set up in order to exhibit the best possible performance, even when dealing only with source-to-source transformations.

1.1.1 Instruction Wall

The instruction wall represents the very first wall that have been bypasses to obtain better performance, i.e. the instruction sequential execution. At the very beginning, the scalar computer executed only one instruction at a time, as shown in Fig.1.2, and had to wait until the first instruction were completely done before starting the second instruction. With this execution pattern, performance improvement can only be achieved through a faster instruction execution.

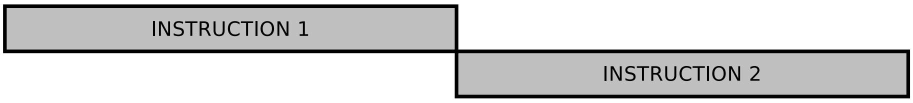


FIGURE 1.2 – Instructions executions on an original scalar processor

When executing an operation, the scalar processor performs the following steps : fetch the instruction from memory, decode the instruction to understand the operation having to take place, execute the operation then store the result of the instruction. All instructions do not require all the same elements, as some instructions will write the result in register or in memory, or on more advanced computers some will require integer processing unit while other will need the floating-point processing unit. From these observations, different mechanisms were designed to improve the instruction throughput. The execution divided in several stages inspired the instruction pipeline, then the diversity of instructions combined with the instruction pipeline lead to superscalar processors, VLIW processors and finally vector processors.

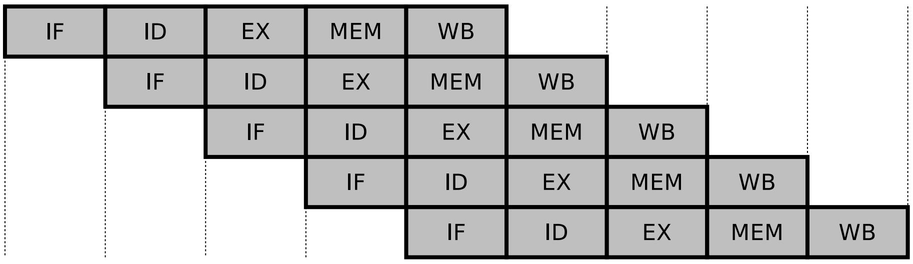
Instruction Pipeline

The idea of the instruction pipeline is analog to an assembly line. Since the sequential steps are independent, why do not we do several product in parallel, with each product in a different stage. An instruction execution being intrinsically decomposed in several step, realizing an instruction pipeline consists in separating the execution of each stage. Even if a simple version was used in 1939 for the **Z1** computer and later in 1941 for the **Z3**, the **IBM Stretch** project in 1961 [39] and the **Illiac II** project in 1962 [20] are considered to be the firsts pipelined processors.

Fig.1.3 displays the classic RISC (reduced instruction set computer) pipeline divided in five stages :

1. **IF** : Instruction Fetch. This step gets the instruction from memory.
2. **ID** : Instruction Decode. This step decode the instruction to signal the required elements, and fetch the date in the correct registers.
3. **EX** : Execute. The operation is done during this step.
4. **MEM** : Memory access. The memory is accessed at the right address. If data is supposed to be written in memory, it is done here. If a data is supposed to be read from the memory, it is done in the next step.
5. **WB** : register Write Back. Result of operation or loaded data is written back in register.

The ideal throughput of a pipelined processor is one instruction per slower stage latency. However, an instruction pipeline is more complex. Some instructions do not



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

FIGURE 1.3 – Instructions executions on a 5-stage pipelined scalar processor

use the **WB** stage, and bypasses are added to skip stages. Also, instructions may depend on data produced by the precedent instruction, and stall mechanisms are designed to wait for the correct data to be available. Moreover if the instruction being executed is a branch, there is no way to know which should be the next instruction to enter the pipeline. However, prediction mechanisms were designed to reduce the overhead of such instructions, filling the pipeline with the most probable choice, and deleting the pipeline content if wrong.

Having instructions stalling induces that several instructions may require the same resource. An instruction must thus wait for the resource to be free. This case inspired superscalar processors presented in the next paragraph, in which the pipeline is widened to the most problematic stages (mainly execution and memory stages) for more operations to take place during these stages.

Nowadays, instruction pipelines are wider than the small classic RISC five stages pipeline. Considering only intel processors in the last decades, the **Pentium 4** architecture displayed a pipeline with twenty stages. Such processors are qualified as "superpipelined", with thirty-one stages reached by Intel's **Prescott** microarchitecture.

Superscalar Processor

As said before, the idea of the superscalar processor is to have multiple instances of the same resources available for instruction executions. If, due to stalls, multiple instructions pass through the same stages simultaneously, all the instructions can operate this stage without bothering another one.

Seymour Cray's **CDC 6600**[96], build in 1964, is often mentioned as the first superscalar design. With performance at 1 Mflops, three times better than the previous top processor, it is considered to be the first successful supercomputer, staying on top of the game for five years. It was dethroned by its own sucessor, the **CDC 7600** [123].

Fig.1.4 displayed an extreme form of superscalar processor, where all stages were widened for two independent instructions to be done in parallel.

From this extreme example, one can extrapolate the principle of VLIW processors. Instead of having several pipeline in parallel executing separate instructions, these instruction are packed in a very long instruction word. The long instruction is fetched, all embedded instructions are decoded and can be executed in parallel.

Another derivation from this model leads to vector processors. Instead of having multiple instructions operating simultaneously on multiple date (MIMD), one can execute a single instruction, operating directly on multiple packed data (SIMD), with enough arithmetic and memory resources not to stall.

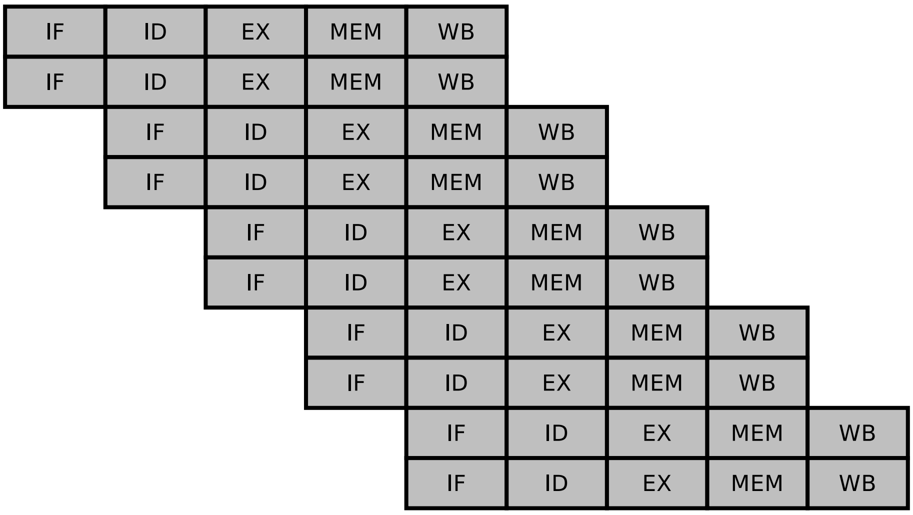


FIGURE 1.4 – Instructions executions on a superscalar processor

Very Long Instruction Word Processor (VLIW)

The first VLIW compiler was described in a Ph.D. thesis written by John Ellis and supervised by Josh Fisher [44]. A very long instruction word is composed with a certain number of short instructions, which are fetched simultaneously, and can be executed in parallel, provided there is no dependency between two instructions embedded in a long instruction. An example of VLIW processor packing four short instructions is schematized in Fig.1.5.

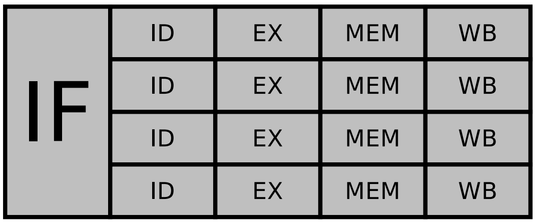


FIGURE 1.5 – Instructions executions on a 4-instructions VLIW processor

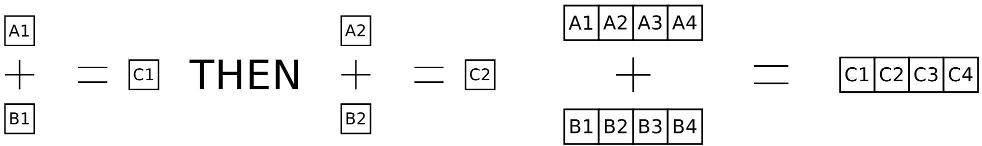
The long instruction is built such as no resource conflict occurs between its own short instructions. For example, the Intel's **Itanium 2** processor can execute three instructions packed together in a *bundle*, which exhibits building constraints. One of these constraints is the impossibility to have three memory instruction in the same *bundle*. These constraints ensure that there will be no stall due to unavailability of resources in the same long instruction. Vector processor will not have this problem, since only one instruction is issued per clock tick.

Vector Processors

Vector processors are processors handling vector instructions, i.e. instructions operating on packed data, comparable to arrays of values. The same operation is applied on all values of the array(s), like for the vectorial add in Fig.1.6.

Specialized instructions are used when performing vector computation. Data have to be loaded in vector registers, and the result is also stored in a vector register.

Vector processors first appeared in the 1970's, the vector techniques being fully



(a) Sequential Add

(b) Vectorial Add with four elements

FIGURE 1.6 – Difference of execution between a sequential Add and a vectorial add with vectors of 4 elements.

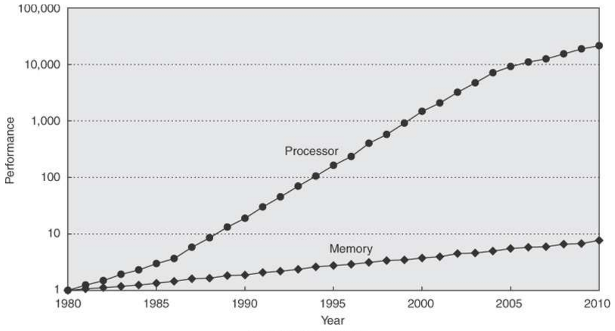
exploited first by the **CRAY-1** computer [90] and the Control data **STAR-100** computer [86]. Vector processors formed the basis of most supercomputers since 1980's.

The hardware constraints and usage of vector computation on current architectures will be discussed in Section 3.2.2.

To continue to break the instruction wall, other hardware features were designed, such as hyperthreading running several instruction streams on the same core, or out-of-order reorganizing on the fly incoming instructions. The formidable constant growth in available performance has led to the memory wall. Performance of memory accesses and data transfers did not increase as fast as computational capability. Hence for most of real codes, the main issue shifted from doing more operations to getting enough data to operate on.

1.1.2 Memory Wall

The "memory wall" is the result of the growing disparity of speed between CPU and the memory outsize the CPU chip. With a constant focus on computing efficiency, and a lower cost of components, CPU speed faster than the memory speed. This issue began to show up in the late 80's, early 90's, as shown in Fig.1.7.



© 2007 Elsevier, Inc. All rights reserved.

FIGURE 1.7 – Processor-memory performance gap through the last three decades

The "memory wall" was formally described, and named, in 1995 by Wulf and McKee [118]. Memory hierarchy tried to hide this gap, multiplying memory layers of small latency between the CPU and the global memory. The purpose was twofold : limiting the access to the high latency memory by keeping in closer memory a maximum of loaded data, and load data from global memory to the closer ones while the CPU works on data in the closer memories. The multiplication of cache levels in modern CPU since the last two decades is an example of such designs.

Caches

Caches play the role of buffers between the processor and the memory (RAM). When a data is loaded from memory to be used by the processing unit, the data is copied in cache. Data in cache can be reused without accessing the global memory. Cache memory, either external or integrated directly into the CPU die is faster than the RAM memory. Due to the cost of such fast memory and the issue of heat dissipation, caches are smaller than the global memory, and the closer to the cpu, the faster and smaller it is. A sort must be done to select which data should stay in cache and which one are no longer useful, and several strategies are implemented to deal with cached data, such as the Round-Robin replacement for a cache store [55].

Using multiple memory levels closer to the processing unit is not new, and the first documented use of a data cache is on the IBM **System/360 Model 85** [28]. With the development of the technology, several distinct caches on the same level were used, stocking either the data or the instructions, and a third translation look-aside buffer (TLB) to speed-up the virtual-to-physical address mapping.

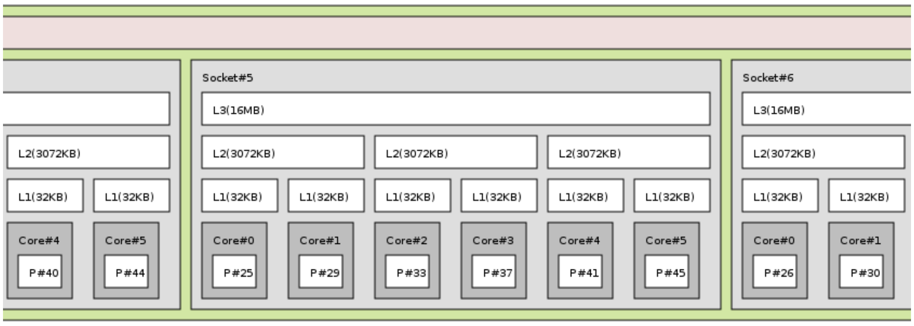


FIGURE 1.8 – Example of memory layout with caches on a multi-core chip

With the wide gap between computing performance and data transfers, it is paramount to use efficiently these fast memory. Programmers have to think their algorithms and implementations in consequence, maximizing the temporal locality (applying several operations on the same data to keep it close) and spacial locality (packing operations requiring the same data, as in an array swap). Also during intensive computation, it is possible to bring data that will be used in cache, hiding the memory latency with computation.

Prefetching

The memory prefetch consists in bringing a data from a high memory hierarchy to a closer hierarchy. This method is mainly used with non-blocking memory access. A blocking memory access consists to wait until the memory instruction is answered before continuing the program execution. For example, if a load is issued and the data is in RAM, one has to wait until the data is loaded in register before doing the next instruction, even if the loaded data will not be used immediately in computation. To be able to hide some memory latency, non-blocking memory access were designed. When using these instructions, the execution does not stall when the data is sought, but it is actually required for a computation.

Thus, it is possible to prefetch data from RAM to a low level cache while doing computations requiring other data. It can be very efficient in loop nests since it is possible to execute the current iteration while loading data for the next iteration.

However, prefetching data can occur several conflicts. First, on some architecture like Intel Core 2 architecture, a hardware prefetcher is integrated, and already try automatically to bring data closer to the CPU. A software prefetch done by the programmer may interact badly with it, and bring only additional overhead instead of reducing latencies. Furthermore, the cost of maintaining cache coherency is increasing dramatically with the multiplication of same level caches required to support larger multicore chips. One has to be sure where the prefetched data will be required to not solicit more than necessary coherency protocol.

Cache Coherency issues

Due to the multiplication of caches on a single chip, cache coherency protocol became heavier and more complex. Considering one of the chip represented in Fig.1.8, one can find four L1 and L2 caches, private to the corresponding core, then a L3 caches shared between all the cores. Clearly, finding the valid required data is more difficult than with previous processors. With a mono-core processors, if the data were not in a memory level, then it was sufficient to ascend through all levels until finding it. Now, with all these caches, the data required by a core can be found in the corresponding L1 cache, but not be valid, the latest version being in another core's L1 cache. When modifying a variable, it may be necessary to broadcast the information that the data are modified, and current copies in other separate caches are not valid anymore.

The complex cache coherency can be maintained with several protocols, one of them being the MESI protocol described next. The MESI protocol was developed at the University of Illinois at Urbana-Champain (UIUC), and the name comes from the first letter of each different states marking the cache lines :

- **Modified** : The cache line is present only in the current cache, and the data is different from the original copy in main memory (the data has been locally modified).
- **Exclusive** : The cache line is present only in the current cache line, and is a copy of the value in main memory (either the data has not been modified, or the new value has already been written in memory).
- **Shared** : The cache line is also present in other caches, and can be invalidate at any time by the other caches. Data matches those in memory.
- **Invalid** : The cache line is invalid. It is not the latest version of the data, and valid cache line must be reloaded from higher memory hierarchy.

Such protocols require more circuit, and generate message passing independent from the running program, and invisible to the user. In symptomatic cases, the additional messages can slowdown the overall execution, whereas the user wrote its program to fit in lower caches to avoid the main memory latency.

The complexity of caches hierarchy induce more power consumption and heat generation, which are the first causes of the third wall outlined in Section 1.1.3.

With the apparition of specialized hardware accelerators, caches management was abandoned in favor of simpler memory model. Those "scratchpad" memories are less automatically handled, and the user manages himself all the memory access, being the sole accountant for the validity (or invalidity) of the data he uses.

Scratchpad

The scratchpad memory is a fast internal memory with no coherency protocol integrated. It uses specific instructions to move data from and to the main memory, implying that data in the scratchpad are not necessary also present in the main memory. If such memories were early considered as additional memory for temporary accesses [92] or a good alternative to cache designs in embedded systems [9], scratchpad memory seems to be the solution for the memory energy consumption on hardware accelerators. SPE's local stores in the late CELL BE, and the local memory in NVIDIA GPUs are example of the choice of scratchpad memory over cache designs (even if the latest NVIDIA card integrate a L1 cache for its processing units). Since efficiently programming accelerators already requires specific insight of the architecture, especially transferring the data from the host to the device, or simply between two chips, completely managing memory transfers for these machines does not induce a large learning overhead. To facilitate the usage of this simple memory design, works have been done to manage automatically data movement at compile time, instead of exhibiting all transfers [68].

Shifting the memory management from the hardware to the software reduces the power consumption of the chip, inducing less heat generation, which is the key to maintain the increase of performance despite the thermal wall.

1.1.3 Thermal Wall

The thermal wall hit in the early 21^{st} century is simply due to a physical issue. The heat generated by the circuit power consumption was to high, and increasing the density of transistors on a chip could not offer anymore a heat dissipation rate sufficient enough. This wall is said to have caused a paradigm shift in the microprocessor industry. But from a certain point of view, it really caused two paradigm shifts, happening nearly simultaneously. The first one is the change from mono-core to multi-core chips, resulting in having to consider parallel programming model instead of the historic sequential one. To deal with the heat density and the power constraints, the industry went from complex and high clock rate single core chips to designing chips with multiple simpler and lower frequency cores cores. The second paradigm change is, coupling the thermal with the processing wall, the appearance of simpler and specialized hardware accelerators, such as GPUs, or SPEs in the late Cell BE. To do more computations in a smaller amount of time lead to the creation of specialized devices being able to do some specific computations very quickly. Most of the hardware features not related to these specific computations are removed, ending up with a simpler and less consuming chip, but with most of the remaining features less automatically handled. The user must have a better insight of the hardware mechanisms to use it efficiently and to obtain the promised speed-up.

Chip MultiProcessor

A multi-core processor is a single computing chip with two or more actual processing elements. The concentration of transistors generating too much heat to dissipate for the processor integrity, vendors started duplicating processors on the same die. Having several distinct processor on the same chip allows to continue increasing the overall peak performance of the chip bypassing the thermal problem. In the last decade, each vendors designed their own multi-core chip. Let us mention among others, IBM's **Power4** and successors, Intel's **Core** duo and **Core 2** duo, and Sun's **Niagara** architectures. If the firsts designed were crude, with two very distinct processors with all resources private, rapidly the subsequent chips took advantage of the shared resources, with the multiplication of shared level of caches for instance (see Fig.1.8). If such common resources may allow a better data sharing, with faster transfer from one core to another, it may also introduce new problems related to waste of shared resources, or contention when trying to access these resources.

To reach the peak performance of a chip is more complex with a mutli-core system. With a mono-core processor, one had just to determine if the running program was memory bound or compute bound. For example, if it was memory bound, the performance would be limited by the memory bandwidth. Now, one has to consider not only the limiting instructions on one core, but also resources interactions across the multiple cores on the chip to achieve its peak performance. Using efficiently the possibilities of a multi-core chip is not straightforward, having to care about how the resources are shared, where the data are placed and will be needed, the memory contention ; all these new constraints merging with the pre-existing onces, such as alignment for vectorization, blocking for data reuse, or covering data latency with computations. Programming for high performance computing becomes more complex with multi-core chips adding new levels in the overall computational and memory hierarchy of already complex distributed systems. And those constraints consideration will increase furthermore with multi-core chips shifting to many-core chips, having not only several processors, but tens to hundreds of processors on a single die.

Hardware accelerators

Hardware accelerator is the use of specialized hardware to perform come operations faster (or cheaper) than it is possible when using a general-purpose CPU. Inspired by the thermal wall, hardware accelerators are circuitry specialized for specific computations. These accelerators are stripped from complex features, presenting only those necessary for instruction executions, and control management. Due to their simpler design, they consume less power than a CPU while performing their specialized operations faster, or at least at the same rate, than the CPU. However, the simplicity of the architecture is only matched by its complexity to program. As management features are missing, the programmer has often to understand what is missing, and considering it directly in its program. The memory is a good example of its behaviour ; memory management and cache coherency is a huge power consumer, most hardware accelerators favored the use of scratchpad memory, with which the user must explicit all memory transfer, and is the only voucher for the validity of data being accessed in the shared memory.

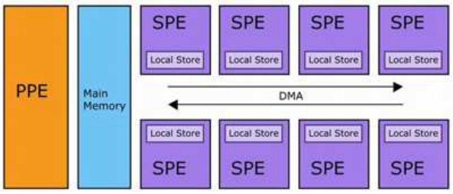


Figure 1.9 – schematic representation of a Cell BE with its PPE and its eight SPEs

The hardware accelerators may be embedded on the chip, as the SPEs of a Cell Broadband Engine achitecture (Cell BE, Fig. 1.9), or external hardware add-on like General-Purpose Graphics Processing Unit (GPU,

Fig.1.10. For the Cell BE, each chip contains the main processor, named PPE for Power Processing Element, and eight accelerators, named SPE for Synergistic

Processing Elements. An Element Interconnect Bus connects all the on-chip elements (PPE, SPEs, the memory controller, two off-chip I/O interfaces). The PPE, act as a controller for the eight SPEs, sending data and instructions to the specialized processing unit. Several Cell BE chips can be linked together to form a larger computing power, with SPEs talking only to their corresponding PPE, and only the PPEs exchanging messages to transfer data among the different chip.

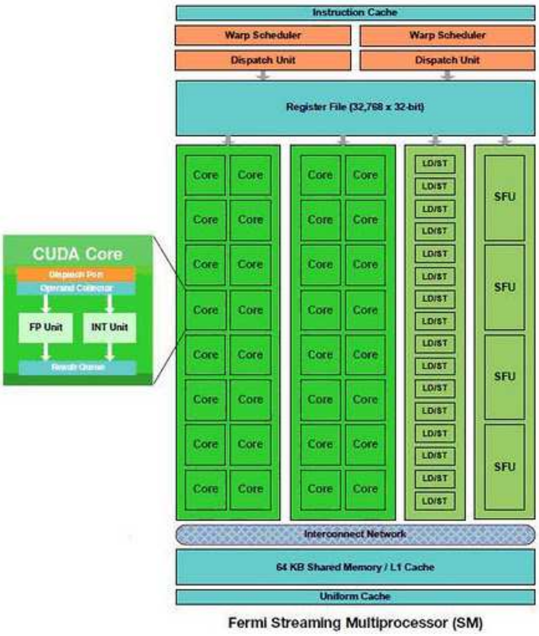


FIGURE 1.10 – schematic representation of a Fermi NVIDIA Card

For the GPU, the machine configuration is not fixed, as there can be one GPU for one or several CPUs, or several GPUs for one CPU. If there are too many GPUs for a CPU to handle, contention issue may occur when transferring data to the device, to the point that time lost transferring data to the GPUs may not be gained using the GPUs for computation ([109]). Considering a NVIDIA Fermi GPU card, it can be composed of one or several Streaming Multiprocessor (SM), which is a module with 32 parallel cores, a shared memory, a large register file, and some scheduler and control units. The processors in different SMs of a chip can access a common global memory, which have a very restrictive way to transfer data efficiently. To be handy, data has first to be send from the CPU host to the global memory, then coalesced data must be moved to the shared memory. A single SM manipulating 32 threads simultaneously, the program must sustain heavy parallelization for the GPU to be efficient.

With hardware accelerators, one can delegate part of a computation to specialized elements, but has to pay attention to all the specificities of used accelerators, along with message passing between devices on the heterogeneous machine. Moreover, using the computational power of the CPU host may increase performance or energy saving, and the programmer must then create two codes taking advantages of each architectures features, and interacting the most efficiently possible. Generating efficient codes for heterogeneous machines in an automatic way is difficult, and is a challenge of this post "thermal wall" age.

Due to the complexity of modern architectures, compilers are struggling to provide efficient codes from a basic implementation. Developers may have to transform the initial code to help the compiler producing an executable taking advantage of the target features. Source-to-source transformations allowing such modifications have been heavily discuss for numerous years. A subset of these transformations, corresponding to the ones used in the main contributions of this dissertation, are presented in the following section.

1.2 Source-to-Source Optimizations

Software optimization is the process of modifying a program to make it more efficient. The focus of this improvement can be time consumption, resource consumption or energy consumption. The increasing complexity of hardware features in modern processors makes the combined effects of a sequence of optimizations hard to predict. Effects of one single optimization depend on both the application and the target architecture. Static compilers resort to simplified performance models, leading most production compilers to trigger some optimizations only in very favorable cases [53, 116, 80]. The works presented in this dissertation rely on source-to-source optimizations to help the compilers find these favorable cases.

Source-to-source optimization allow the user to directly transform the original code in a new version, more efficient or simply closer to the machine architecture. Most of these modifications apply on a loop nest. The transformations used in this dissertation are presented in this section. A more complete and detailed list of code transformations can be found in Sylvain Girbal's Ph.D. Thesis dissertation [50].

1.2.1 Code motion

Code motion moves an instruction block to another location, in the same level of a loop nest. Being moved backward or forward, it can be considered that the block is swapped with its predecessor or following block. To apply code motion, there must be no dependences violation between the swapped blocks.

Code motion is a classical source-to-source transformation analog to a copy-paste manipulation. It is mainly used to bring closer data producers and consumers, improving the temporal locality of variables. In symptomatic cases, code motion can remove the use of temporary variables or arrays.

Code motion can be applied only if no dependencies are violated when moving the block of code.

Below is an example applying some code motion between the two blocks of code (namely block_2 and block_3) on a pseudo-code representation :

```
block_1{                              block_1{
|  ...                                |  ...
}                                     }
block_2{                              block_3{
|  ...                ⟶               |  ...
}                                     }
block_3{                              block_2{
|  ...                                |  ...
}                                     }
```

1.2.2 Extraction

Code extraction (and by extension, loop extraction) consists in taking an instruction block (or a loop) out of a program to put it in a new function. Then this new function is called from the original code to do its part.

Isolating a code portion from the overall program may relaxed constraints for the optimization phase of the compiler [10]. When extracting inner loops of a loop nest, one automatically applies array contraction on the concerned arrays, simplifying the computation representation.

Below is an example of loop extraction applied on the inner loops of a loop nest :

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    loop_body (i, j)
```

⟶

```
for (i=0; i<M; i++)
  function (i; data(i))

function (iterator i, input data_i))
  for (j=0; j<N; j++)
    loop_body_i (j)
```

1.2.3 Tiling

Loop tiling decomposes a linear swap of a loop in several smaller consecutive linear swap. When dealing with a loop nest swapping a multi-dimensional array, it decomposes the considered polyhedral in smaller ones, with constant form and size, to swap.

Loop tiling in a multi-level loop nest is done to change the spacial and temporal locality of data in the arrays, improving data reuse for some computation patterns. To a lesser extent, loop tiling can also be used on a loop with unknown bounds to fix statically the size of loop for the most part of the iterations [111]. Compilers doing a better optimizing job when dealing with statically known loop bounds, global performance will be increased for the tiled loop, with only the reminder loop producing a poor quality code.

Below is an example applied on all loops of a loop nest :

```
for (i=0; i<M; i++)
  loop_body_1 (i)
  for (j=0; j<N; j++)
    loop_body_2 (i, j)
    for (l=0; l<L; l++)
      loop_body_3 (i, j, l)
```

⟶

```
for (i=0; i<M; i+=X)
  for (ii=0; ii<X; ii++)
    loop_body_1 (i+ii)
    for (j=0; j<N; j+=Y)
      for (jj=0; jj<Y; jj++)
        loop_body_2 (i+ii, j+jj)
        for (l=0; l<L; l+=Z)
          for (ll=0; ll<Z; ll++)
            loop_body_3 (i+ii, j+jj, l+ll)
```

1.2.4 Interchange

The Loop permutation transformation exchange the position of two consecutive loop in the same level of a loop nest. Loop interchange is a code transformation very used for code optimization. It can induce a change in the spacial and temporal locality of memory elements in nested loops [4]. Loop interchange may also be used to increase scalar promotion, in order to lighten the memory footprint.

It is possible to apply a permutation between two loops in the same level if there is no dependance between data inside those two loops.

Below is an example of interchange applied in a loop nest.

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    loop_body_1 (i, j)
  for (l=5; l<N; l++)
    loop_body_2 (i, l)
```

⟶

```
for (i=0; i<M; i++)
  for (l=5; l<N; l++)
    loop_body_2 (i, l)
  for (j=0; j<N; j++)
    loop_body_1 (i, j)
```

1.2.5 Fission

Loop fission transformed a single loop in two distinct loops, with instructions in the original loop body being divided between the two new entities. The new loop has the same iteration domain than the original loop.

This transformation is used to separate large loop in several smaller ones. These smaller loops can be better handled by compilers, or reorganized to exhibit better data locality [72]. If the fission separates independent data streams, the only overhead of the fission will be the multiplication of branch instructions. However, is the distributed instructions are part of the same stream, array privatization may take place, adding a new dimension in considered arrays to carry data between the loops. If the original computation is memory bound, it can be very harmful to the overall performance.

Below is an example of loop fission applied on the middle loop of the represented loop nest :

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        loop_body_part1 (i, j)
        loop_body_part2 (i, j)
```

⟶

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        loop_body_part1 (i, j)
    for (j=0; j<N; j++)
        loop_body_part2 (i, j)
```

1.2.6 Fusion

Loop fusion realizes the opposite transformation of Loop fission. It takes two consecutive loops on the same level and with the same iteration domains, and combine their bodies in one new loop, if there is no dependences violation.

Loop fusion allows a different spacial and temporal locality for data in the original distinct loops. In some cases, this new locality may create opportunities for scalar promotion (since the producer and consumer are in the same loop, a simple variable is enough to carry the data, when an array was needed with two loops) [?]. For multi-dimensional array scalar promotion is extremely rare, but array contraction can occur, removing in the arrays the dimension relative to the fused loop level. Even if the created loop is larger, the use of simpler data structure will ease the compilers work.

Below is an example of loop fusion applied on the middle loops of a loop nest :

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        loop_body_part1 (i, j)
    for (j=0; j<N; j++)
        loop_body_part2 (i, j)
```

⟶

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        loop_body_part1 (i, j)
        loop_body_part2 (i, j)
```

1.2.7 Unroll

A full loop unroll duplicates the loop body as many times as the number of iterations, in order to completely remove the loop call. A partial unroll consists in applying first a tiling on the loop, then fully unroll the inner tiling loop.

The first purpose of unrolling is to remove branches instructions compared to computation instructions, reducing the overhead brought by waiting for the branch result. A less obvious optimization of a loop unroll is to enlarge the loop body. As explained before, compilers better handle loops which are not to large. However, too few instructions in a loop body is also a limiting factor for the optimizing routine of compilers. By unrolling small loops, one gives the compiler more material to work on, and may produce a better code in the end. Moreover, unrolling allows to exhibit or improve software pipelining [21, 22].

Below is an example of a partial unroll applied on the inner loop of this simple loop nest :

```
for (i=0; i<M; i++)                      for (i=0; i<M; i++)
  for (j=0; j<N; j++)          ⟶           for (j=0; j<N-1; j+=2)
    loop_body_1 (i, j)                       loop_body_1 (i, j)
                                             loop_body_1 (i, j+1)
```

1.2.8 Skewing

Loop skewing changes the iteration space, replacing loop bounds and iterators with new ones, swapping the same elements in a different order, or even in the same order.

Loop skewing is done mainly to change the reference frame, allowing inner loop parallelization when dependences were not originally parallel to the axis. After the skewing, the inner loop can be cut in several parallel stream without severing data dependences [115]. To a lesser extent, loop skewing may also simplify array indexes evaluations, which can be a real stall in the computation if too complex.

Below is an example of a skewing applied on the presented loop nest :

```
for (i=0; i<M; i++)                      for (k=0; k<M; k++)
  loop_body_1 (i)                          loop_body_1  (k)
  for (j=0; j<N; j++)          ⟶           for (l=k*N; l<(k+1)*N; l++)
    loop_body_2 (i, i*N+j)                   loop_body_2 (k, l)
```

1.2.9 Vectorization

Vectorization consists in hiding several similar computations on consecutive values behind only one call of the computation. The operation applies on vectors, that are like small arrays of consecutive data. If it can be used to represent specific data structures to work on, like for a pixel represented by an array of four int, its effectiveness and the speed-up it can bring is mainly due the hardware circuitry performing vectorial operations on nowadays architecture [79, 15]. When using the vector circuitry, one can perform an operation on several data while the scalar unit would have perform only one operation in the same time. The best performance when using vectorization are achieved when filling the requirement issued by hardware supports for vectorized computation. The hardware mechanisms for valid and efficient vectorization are discussed in Section 3.2.2.

Below is an example of a vectorization, with vector composed of four elements, applied on the inner loop of this simple loop nest :

```
for (i=0; i<M; i++)                      for (i=0; i<M; i++)
  S1 (i)                                   S1 (i)
  for (j=0; j<N; j++)          ⟶           for (j=0; j<N-3; j+=4)
    S2 (i, j)                                S2 (i, {j, j+1, j+2, j+3})
    S3 (i, j)                                S3 (i, {j, j+1, j+2, j+3})
```

1.2.10 Composition of transformations

Code optimizations require most of the time to combine several transformations. Having a good knowledge about data structures, target architectures and optimizations requirement for validity, one can choose, and apply, a specific set of transformations he knows to be the best candidates to achieve high performance.

However, for someone outside such knowledge, it is very hard find the best sequence of optimizations. Moreover, given a set of optimizations, it is not possible to verify the optimality of a sequence of transformations [98].

To apply successive optimizations on an original implementation, one can resort on iterative compilation. It consists in applying first a modification on the code, then to compile and test the modified solution. Then a second modification is done, and so on. To avoid the combinatorial effects of trying all possible recompositions using the given subset of optimizations, several intermediate representations through polyhedral models have been proposed [29, 27, 51, 13, 7, 88]. These models allow to explore multiple optimization compositions, without having to produce the new version at each step. From the representation, some resulting versions can be discarded, whether presenting invalid or bad characteristics . Only the selected sequences of optimizations will be exported from the intermediate representation to an actual implementation.

Even when using polyhedral model, the search for efficient transformation sequences must be guided, either by the user (the solution we choose in Chapter 2) or with heuristics pruning the optimization space.

To complicate even more this search, the performance of some transformations not only comes from the code characteristics, but also from the hardware circuitry supporting their implementation. Vectorization is one of these, with specific registers and computing units dedicated to its physical execution. This hardware support may induce special requirements to exhibit the best performance. The specific case of vectorization is detailed in the following section to introduce alignment conflicts notions used in Chapter 3.

1.3 Hardware support for vector code

On a hardware level, vector registers have a fixed size, and contain as many values as necessary to fill them. A computation on two vectors will apply the operation on each atomic value at the same position in the vector. For example, an `add` operation on two vectors `V1={a1,b1,c1,d1}` and `V2={a2,b2,c2,d2}` will perform `a3=a1+a2`, `b3=b1+b2`, `c3=c1+c2` and `d3=d1+d2`, and store the results in a new vector `V3={a3,b3,c3,d3}`. Loading data into vector registers, enough consecutive data are extracted from memory to fill one. The size of a vector register depends on the implementation for the architecture : 128 bits for Intel's and AMD's SSE, expanded to 256 bits for new AVX registers, 128 bits for IBM's Altivec, 32 bits for NVIDIA CUDA. Hardware mechanisms to fill the vector registers are discussed in Section 1.3.1 for CPUs and in Section 1.3.2 for GPUs.

1.3.1 On CPUs

Filling a register with data from memory may have different performance if the first data to be loaded in the vector has an address being a multiply of the vector size or not. If the address of the first accessed data is null modulo the vector size, the data is said to be *aligned*, and an *aligned* load can be used. SSE aligned load of simple precision values are represented in Fig. 1.11, and aligned load of double precision values in Fig. 1.12.

While Altivec does not support unaligned access, SSE instructions exist to load non-naturally aligned data directly into registers, as long as the alignment requirement of its base type is fulfilled. An unaligned load to create a vector of four simple precision floating point values can take place if the first data is aligned on a 16 bits boundary, but not on a 8 bits or a 13 bits boundary for example. As displayed in Fig. 1.13, the four consecutive data are directly put in the register like

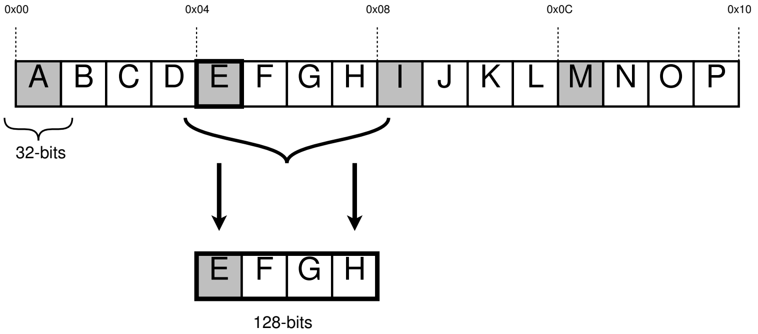


FIGURE 1.11 – Aligned load from memory to a vector register in simple precision

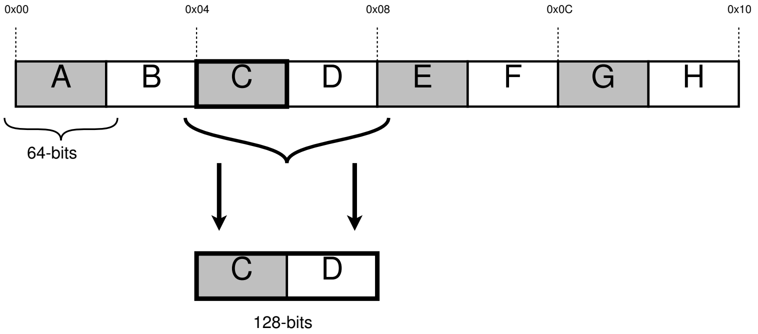


FIGURE 1.12 – Aligned load from memory to a vector register in simple precision

an aligned load. However, more complex internal mechanisms are required to fetch data from anywhere in memory, and an unaligned load has a higher latency than the aligned one. Even if the gap of performance between the two are shrinking, alignment conflicts remain a a key point to achive performance [52] .

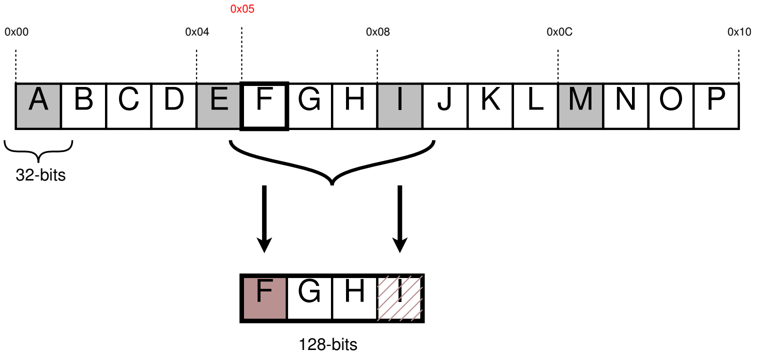


FIGURE 1.13 – Unaligned load from memory to a vector register in simple precision

Another possibility to realize vectorization from unaligned data is to fill multiple vectors with aligned loads, shuffling data inside the vectors then merging them to build the vector actually required for the computation. The complete procedure is detailed in Fig. 1.14, also with SSE instructions, all the step are not necessarily separated, and only one instruction inserted after the two aligned loads may be enough to build the wanted vector.

Vector computations are usually used on the inner loop of a loop nest. Vec-

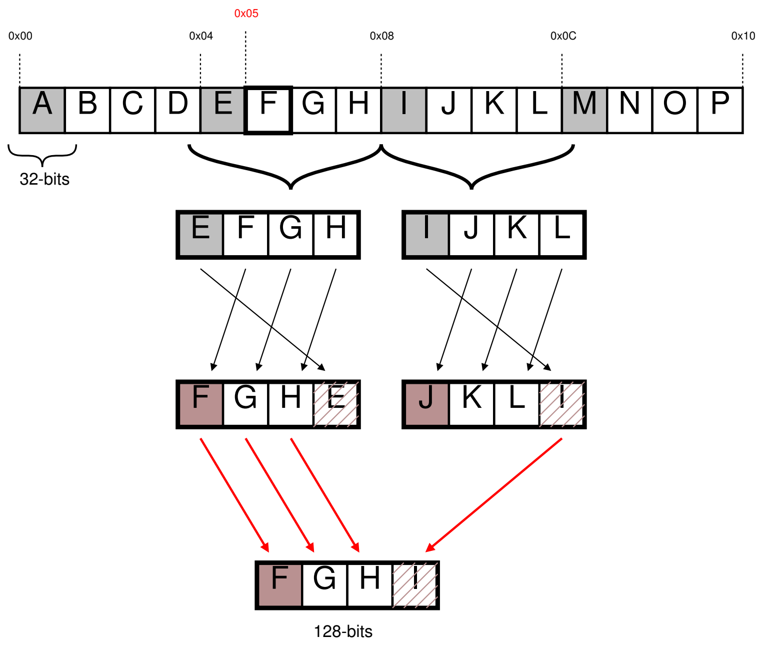


FIGURE 1.14 – Two aligned loads followed by a shuffle operation to build the required vector

torizing consecutive unaligned data, only one aligned load might be necessary to build the required vectors, reuse the already available ones. For IBM's Altivec, the shuffle methodology is the only way to work is unaligned. For SSE instructions, the best choice between unaligned loads and shuffles depends on the program limitations, with shuffling increasing the register pressure and the number of instructions possibly being more time consuming than unaligned loads.

1.3.2 On GPUs

GPU cards from different vendors don't have the same hardware representation of vectors. On AMD cards, it is based on the same model as the SSE representation, and the hardware can manipulate 128 bits vector registers as described for CPUs. On the other hand, there is no specific vector register on NVIDIA card. Only 32 bits registers can be found, and data is stored in the number of registers necessary to handle the value. For example, a simple precision floating point value will be loaded in one register (see Fig. 1.15, while a double precision floating point value will be loaded in two distinct registers coupled to handle the 64 bits value.

If the behaviour of registers seem to avoid alignment issues while loading values into 32 bits registers, it is not the case when using built-in types of the CUDA programming language. Built-in vector types were created to provide programmers with an existing set of vectors, grouping of each types being included, up to 128 bits or a vector of four elements. Vectors of char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float and double are represented, composed with a maximum of four elements for smaller data types (char1, char2, char3, char4), and a maximum of two elements for larger data types (double1, double2).

These built-in types force specific alignment of data, sometimes with vector requirements being different from its base type requirement. Among others, a char1 must be aligned on a 8 bits address, char2 on a 16 bits, char3 on a8 bits address, char

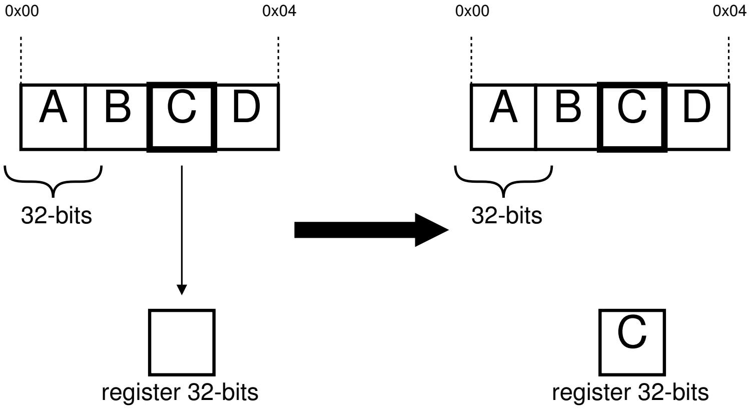


FIGURE 1.15 – Load of a 32-bits data from memory to registers

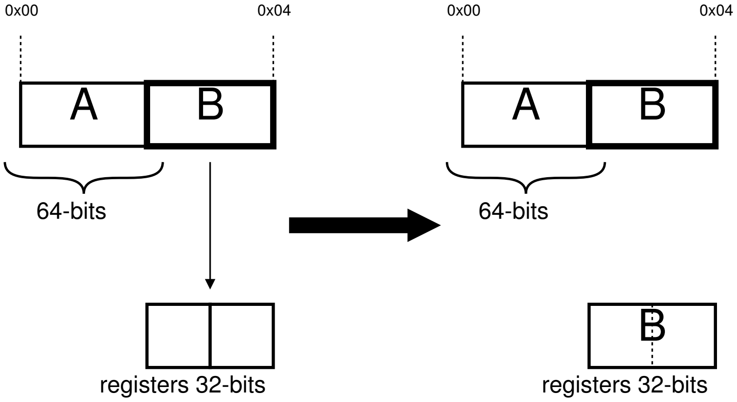


FIGURE 1.16 – Load of a 64-bits data from memory to registers

4 on a 32 bits address, int1 on a 32 bits address and double1 on a 64 bits address (the table for alignment requirements can be found in the Cuda Programming Guide [31]). Using these built-in types will cause the same alignment issues than for CPU.

Chapitre 2

CPC : an autotuning framework for irregular codes

In this chapter is presented a autotuning framework to optimize irregular codes. This framework, called *CPC framework* for *Constant Performance Codelet*, aims to optimize large programs focusing only on their hot-spot functions with nested loops. To avoid the large amount of time spent in iterative compilation to optimize those programs, the CPC framework extract the hot-spot function, slice the loop nest in several part named *codelets*, and optimize only these small codelets. A performance prediction model selects the codelets to recompose an optimized function. The framework described in this chapter have been presented in the SMART'09 workshop [54].

2.1 Introduction

The increasing complexity of hardware features in modern processors makes compilation for high performance very challenging. Effects of one single optimization depend on both the application or the target architecture, and are hard to predict. Static compilers resort to simplified performance models, focusing on one metric (such as cache misses for instance) assessing the assumed effectiveness of the optimization. The combined effects of a sequence of optimizations is therefore most often unpredictable and prevents compilers from finding most effective optimization sequences. Actually, the risk of using an approximate performance model is as much to degrade performance as to miss optimization opportunities. This leads most production compilers to trigger some optimizations only in very favorable cases (xlc compiler and vectorization for BlueGene for instance [53]).

One approach to deal with the problem of simplistic performance model is to rely on performance measures, making code execution part of the compilation time. Adaptive compilation drives selection of optimization sequences by feed-back performance evaluation [59]. To limit the search among possible optimization sequences, several techniques have been proposed[62] : adaptive compilers resort to genetic algorithms [30], machine learning algorithms [1] and some amount of enumeration. Auto-tuning libraries also use adaptive compilation techniques (Spiral [87], FFTW [47], ATLAS [114], Stapl [75] for instance) and have shown their effectiveness even compared to hand-tuned codes. With the exception of BLAS library where it has been shown that a pure performance model approach for ATLAS can be competitive with the empirical search version [120], the main drawback of iterative compilation is its large compilation time. As for each change of compiler flag, optimization parameter or sequence, the whole code is executed. Most of the research on iterative compilation has focused on limiting the search, and consecutively, the total execution time (see the work of Dunwell *et al.* [48] for instance, based on the different phases of an execution). However, these approaches assume some properties on dynamic execution as : the input can be transcribed in a specific representation, the run of the test is repeatable. Our method is based only on static decomposition of the code based on control (and dataflow) properties.

We proposed a novel approach for adaptive compilation, combining empirical search to a performance model. The main contribution is to describe a method that searches for many code variants and predicts performance accurately while only small code fragments are executed. The method is threefold : (i) source to source transformation sequences are explored according to user-defined pragmas, generating as many code versions, (ii) for each version, some inner loops, named codelets, are taken out of the application and executed, (iii) performance evaluation of each code version is obtained by combining codelet timing measures and a simple performance model. This approach is an extension of the method presented in Banakar *et al.* [10] for library generation of linear algebra codes, and we generalize it so that real applications can be targeted. Detailed use of this method, referenced later as the CPC framework (*Constant Performance Codelet*) is described on three "real life" applications : a molecular simulation (section 2.3), a Lattice QCD simulation (section 2.5) and a genomic simulation (section2.4). Experimental results on an Itanium 2 architecture show the performance model is accurate, and that substantial speed-up can be reached benchmarking only parts of the hot-spot functions.

2.2 CPC framework

Simulation codes are generally programs running for a long time. When optimizing such codes, one has to run each new versions to see if the applied transformation is efficient or not. Testing only a few possibilities of optimization will already take a great amount of time, the time lost being useless if none of these optimizations improved the performance. The CPC framework aims to greatly reduce the time consumed in testing the transformations, by selecting only specific parts of the code to optimize. Profiling the original code, the more time consuming function is selected to search for codelets. A codelet must have specific characteristics recognizing it as a *Constant Performance Codelet* (section 2.2.1). If no such codelet is found, we realize some preliminary code transformation to end up with valid codelets to extract, as explained in section 2.2.3. The obtained codelets are then modified through a wide range of optimizations in section 2.2.5. After evaluation (section 2.2.6), the best codelets are used, being called by the hot-spot function in order to recompose it (section 2.2.7) following the memory model described in section 2.2.2.

2.2.1 Constant Performance Codelet

The principle of the method is to explore many code optimizations on codelets, and to predict the performance of a code version, evaluating the codelets composing this version. The time to run a simulation program can be very long. Once the nested loop fits the criteria of a *Constant Performance Codelet*, one can extract it from the original code, in order to optimize it without executing the whole code. We describe in this section the characteristics imposed on these codelets to base the method on a simple performance model.

A general formulation of performance would depend on program inputs and on initial machine state. However, we do not try to model code performance expressed as a function of program inputs. It means in particular that we will not try to predict performance by extrapolation. Performance of codelets are measured in some conditions that will be reproduced in the application so that the measure is reliable. Note that we only focus on cycle counts.

In our performance model, we consider that some code features to be difficult to predict. The *Constant Performance Codelet* will exhibit some characteristics, so the effects of these code features can be either efficiently measured or neglected. These characteristics are verified on assembly code (this may differ with source code) :

1. The codelet is a loop (at least one loop). For large enough loops, the codelet execution takes enough cycles so that impact of the instructions before and after the codelet (through out-of-order mechanism for instance) can be neglected. The execution time of two consecutive codelets will be the same as the sum of their respective stand-alone evaluation time. The "large enough" criteria is architecture dependent. For loops with only a few iterations, our performance prediction may be very different from the real evaluation.

2. Control flow does not depend on input data. It means constant loop bounds, no while loops, and conditionals depending only on loop counters. It reduces effects due to branch wrong prediction. Note that `if..then..else` constructs in the source code may be translated into straight line block of code with conditional instructions, for architectures having such predicated code (Itanium for instance). In this case, as all instructions of both branches are executed, execution time does not depend on the value of the test (as the example of Gibbs in the introduction).

3. Dataflow dependences in the codelet do not depend on inputs. If these dependences depend on inputs, performance prediction will lose accuracy. Due to hardware design, dependence testing (a read after a write for instance) is achieved by comparing only a few bits of the addresses. This may cause stalls, even if there is no real dependence. The number of stalls changes only if relative position of starting addresses of arrays changes. We assume that all arrays are aligned to the same boundaries.

4. Configuration of inputs in cache is known when the codelet starts. This condition, combined to the previous ones, ensures that the misses and hits of the codelet will be exactly the same between two executions. However, this assumption implies that for each different configuration of data in cache, a different performance measure has to be made, thus limiting the number of configuration. In our examples, we considered only the case when data was fully contained in a cache level or not. This will induce inaccuracy in one of our presented results.

5. No I/O or system calls. They can cause variation of performance hard to predict.

These conditions define a *constant performance codelet* and the search aims to decompose the code into these codelets. As a consequence of their definition, these codelets can be taken out of the application context and benchmarked *in vitro* to evaluate their performance in different input configurations (cache hierarchy that contains these inputs). Compared to traditional iterative compilation framework, there is no need to execute the whole application.

Codelets preparation

If no *Constant Performance Codelet* is visible in the hot-spot function, small preparation may be necessary to reveal them. Most of the time, classical loop transformation can remove CPC requirement issues due to control flow. If the loops are while loops, or for loops with no constant bounds, some tiling will add a new level in the loop nest, with fixed bounds (the tile size). This new loop will be eligible to apply the framework. On the other hand, `if..then..else` constructs inside the loop body will be dealt with by applying fissions, separating the conditional blocks from the computational ones. The framework will then consider only loops with no conditionals.

2.2.2 Memory model

We use a simple memory model based on cache capacity. We suppose that data of a codelet completely fill a cache level, or only fit in memory.

When we recompose the function with the best codelets, we have to ensure that the data are in the same memory configuration as the evaluated codelets. If not, the execution time measured while evaluating the codelets will be different from the execution of the codelets in the recomposed function, and the prediction will be wrong.

As the model assume codelets data are in some cache, it requires some cache model to know the data region to move. The objective is not to modify the codelets by adding prefetch or copy instructions inside, as this would alter performance. On the contrary, prefetches and copies will be added by blocks, in the recomposed function, before the corresponding codelets. The prediction will be more accurate with this method. Performance of these blocks are also be measured.

Blocks of copies are used to brought the required data in caches, while changing the disposition of the elements. Blocks of copies will be preferred to blocks of prefetches if the integrity of the data structure of the codelets is altered. As an example, consider a vectorization applied on a 2D array, allocated with only one block of memory. Vectors are filled with contiguous data in memory. If the line size of the logical 2D array is not a multiple of the vector size, data at the end of a line, and at the beginning of the following line, will be aggregated in the same vector. Computations using this vector may produce wrong results, as the vector is not a correct vector, from the logical 2D array point of view. A copy is then necessary to move the codelet data in a new structure where such incorrect behaviour will not occur. The writing of the new structure will automatically put the data in the closest cache level. Blocks of prefetches are used to load in advance the data used by the codelet in the correct cache level, if the codelet was evaluated considering that its data were not in memory.

We need to measure block of copies and of prefetches designed for particular codelets. Let us whether they can be considered as constant performance codelets. They have constant loop bounds, static control, and a loop. Moreover, dataflow dependences inside the codelets do not depend on inputs. Different tests are needed, according to the position of the data to copy/prefetch in the memory hierarchy. However, performance still depends on spacial locality since two data in the same cache line and accessed consecutively will be accessed faster than if they were mapped to two different cache lines. Variation between two executions with different input data (not size) only happen for A[B[i]], due to this effect. When an indirection of the kind A[B[i]] occur in a copy, we choose the worst case execution and initialize B such that it addresses different cache lines.

Evaluating block copy and block prefetch

We need to measure blocks of copies and blocks of prefetches designed for particular codelets. These codelets exhibit the conditions to be considered as constant performance codelets. They have constant loop bounds, static control, and a loop, dataflow dependences inside the codelets do not depend on inputs. Different tests are needed, according to the position of the data to copy/prefetch in the memory hierarchy.

The prefetch instruction is not a blocking instruction, meaning that other instructions can be issued while the memory request is not answered. The first next instruction requiring a data still being prefetched will stall the running code to wait for the data. In good cases, the memory latency of prefetch instructions can be completely hidden with computations. For blocks of prefetches, we design a codelet that prefetches all data and add to it a small computation consuming this data as fast as possible (with a reduction). When the data prefetched is not in cache, an evaluation of this codelet measures the latency due to the miss, and the overhead of the reduction itself. It is important to notice that such case can be considered as a worst case, since data prefetched is accessed right after the prefetch, and the computation can not be used to hide the memory access latency. In an application a more favorable case may occur. Evaluation of a block prefetch will return a maximum for its time consumption. Figure 2.2.b shows the performance of a codelet of prefetch for 15.10^9 elements (spinors) required by a codelet from the HMC application. The left bar measures only the overhead of the codelet, when data is already in cache. The right bar shows the latency of the memory access, when data in memory is prefetched and then used. This latency minus the overhead provides the performance measure (worst case) for prefetches.

Blocks of copies are evaluated the same way, as even memory copy can be non blocking instructions. Here again, their evaluation will return a maximum of time consumption, and their running time in the recomposed function will be lower, or the same time in the worst case.

2.2.3 Codelets extraction

Once the *Constant Performance Codelets* are identified, they are extracted in stand-alone functions. Extracted codelets are considered without any knowledge of the global program environment. Prior to this extraction, some transformations, like fission or fusion, can be applied to increase the number of possible codelets to study. For example, in a loop nest with at least two levels, the innermost loop can be extracted unchanged, or after an interchange. Hence, there will be two possible basic codelets for this loop nest, one based on the innermost loop, the other based on the interchanged one. The larger the number of basic codelets for a loop, the greater the chance to find an efficient optimization.

Scalar promotion and array flattening are transformations inherent to the extraction. When a codelet is extracted, all variables known before the codelet are considered as constant inside the codelet, like loop counters of including loops. An array with two dimensions can have a dimension removed, if the swapping loop is outside the codelet. Scalar promotion will reduce the number of memory accesses inside the codelet. Array flattening can simplify memory accesses, removing indirections from the codelet body. These inherent transformations are already a first optimization of the codelet, as compilers often work better with simpler array structures, with simpler memory access patterns.

The identification and the extraction of the codelets are guided by the programmer. We focused our research on the loops which take the major amount of time in our programs. To automate this part of the work, a simple code extractor should be coupled with a profiler tool to extract only the more interesting loops to optimize, or at least to give some hint to the programmer on which loops the search should be apply. Existing code isolators [67] tend to collect a complete trace of the machine state when executing the codelet in the whole program to offer the same runtime environment for the isolated code. Our method allows the extracted codelet to be independent from the program environment, and, at the end, it will be the program to be adjusted according to the more efficient optimized codelet.

2.2.4 Performance Model

The characteristics of the constant performance codelets, added to the memory model we use, allow us to base our performance prediction on a simple additive model.

Extracted codelets can be considered as new instructions, specific to the application, in the recomposed function. There is no parallelism between these instructions. Since the memory configuration if fixed through copy or prefetch blocks, no time gain can be achieved with the improvement of data locality between successive codelets.

The sequentiality of the codelets, and the impossibility to improve data reuse between them, ensure that the total execution time of the combined codelets is the sum of the evaluated times of the codelets. The only uncertainty comes from the copies and prefetches blocks, which can be partly or totally covered by computing codelets, reducing the impact of these memory codelets. The predicted execution time of the combined codelets is an upper bound of the real performance of the new function.

2.2.5 Optimization space with X-language

The goal of this step is to search among all code transformations those that produce, from an original codelet, a high performance codelet. The optimization space is user-defined. With the help of a specific language designed for this purpose, the user insert in the original code pragmas describing all the optimization to try.

Defining search space with X-language

X-language relies on a C99 compiler `tcc`[85] that passes an AST to a Prolog program that does the effective search, based on a set of rules. X-language is developped at the university of Versailles, and has been used to generate many versions of codes for linear algebra libraries[10]. Our test programs being more complex than basic linear algebra subprograms (or BLAS), we added new transformations in the language to produce efficient codelets for these programs. The transformations added to X-language are :

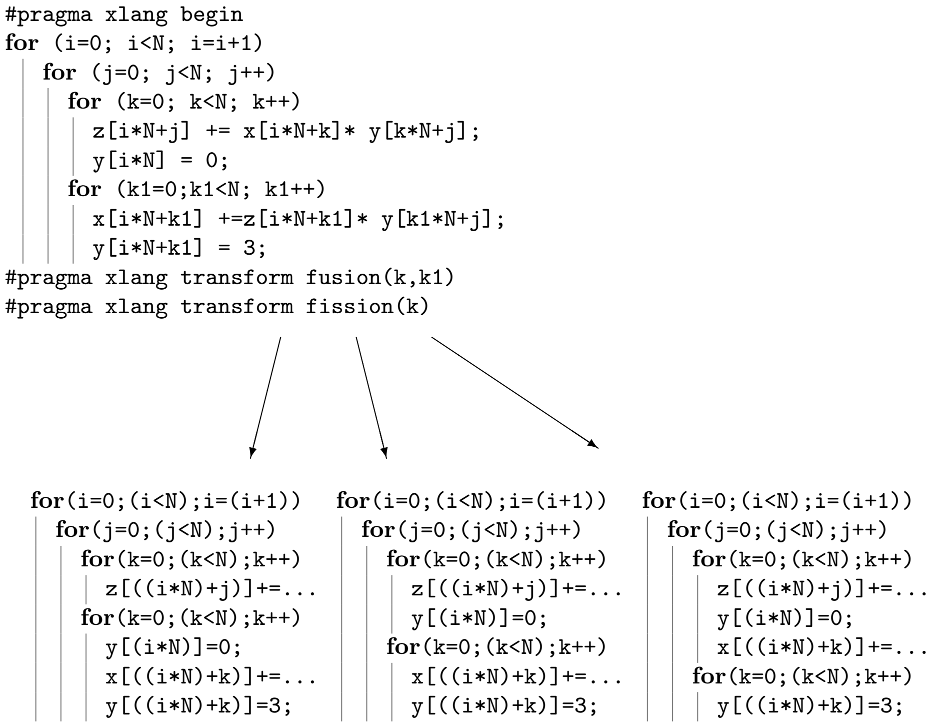
- `#pragma xlang transform unrolljam`($loop{-}id, unroll{-}factor$) : unroll and jam is a simple extension based on existing transformations and dependence analysis provided by previous pragmas. The jam performs fusion of all loop surrounded by the unrolled loop.
- `#pragma xlang transform vectorize`($loop - id$) : vectorization is pattern based and applies only if all statements of a loop can be vectorized. Vectorization is decomposed into as many pattern based rules as there are vector instructions.
- `#pragma xlang transform tryintrinsics`($name$) : it matches all function names and replaces them with the equivalent intrinsics (if present in rules). This directive will generate codes with and without the intrinsics substitution.
- `#pragma xlang transform distribute`($loop{-}id$) : distributes statements in different loops, preserving dependences given by the previous pragmas.

For all applications, two parameters are searched for : tile sizes (generating inner loops with constant loop bounds) and unrolling factors. X-language does not have by itself a dependence analysis. This removes a constraint due to overly conservative dependence analysis, but may also generate incorrect code. For some transformations (such as slicing) it is necessary to indicate the group of statements that must be kept in sequence. This is achieved by two pragmas :

- `#pragma xlang section` id where n is a unique number. This indicates the beginning of sequential section of code.

– #pragma xlang dependence($list - of - ids$). This indicates that a chain of dependence between sections of code.

Here is an example of the X-language program used to explore different versions of a nested loop through fusion and fission.



```
#pragma xlang begin
for (i=0; i<N; i=i+1)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      z[i*N+j] += x[i*N+k]* y[k*N+j];
      y[i*N] = 0;
    for (k1=0;k1<N; k1++)
      x[i*N+k1] +=z[i*N+k1]* y[k1*N+j];
      y[i*N+k1] = 3;
#pragma xlang transform fusion(k,k1)
#pragma xlang transform fission(k)
```

```
for(i=0;(i<N);i=(i+1))
  for(j=0;(j<N);j++)
    for(k=0;(k<N);k++)
      z[((i*N)+j)]+=...
    for(k=0;(k<N);k++)
      y[(i*N)]=0;
      x[((i*N)+k)]+=...
      y[((i*N)+k)]=3;
```

```
for(i=0;(i<N);i=(i+1))
  for(j=0;(j<N);j++)
    for(k=0;(k<N);k++)
      z[((i*N)+j)]+=...
      y[(i*N)]=0;
    for(k=0;(k<N);k++)
      x[((i*N)+k)]+=...
      y[((i*N)+k)]=3;
```

```
for(i=0;(i<N);i=(i+1))
  for(j=0;(j<N);j++)
    for(k=0;(k<N);k++)
      z[((i*N)+j)]+=...
      y[(i*N)]=0;
      x[((i*N)+k)]+=...
    for(k=0;(k<N);k++)
      y[((i*N)+k)]=3;
```

FIGURE 2.1 – Application of X-language on a loop nest to generate several versions resulting of fusion and fission transformations..

X-language allows to precisely define the scope of the transformation space, while providing enough modularity to easily add new transformations when it is required. Some phases of the search, notably the writing of initializing code for the codelet, are still performed by hand.

2.2.6 Codelets evaluation

Before benchmarking the codelets, a first selection is performed through the analysis of the assembly code generated from the codelets. We used a software tool, MAQAO [36], developed at the university of Versailles. The MAQAO tool analyses the assembly code, and gives hints about the quality of the generated code. With these hints, we will decide if the optimized codelet should be discarded, or is a good candidate to benchmark. For example, MAQAO will tell us if the assembly loop body has one basic block or not. If the loop is composed with several basic block, it generally means that conditionals in the original code have not been predicated, or the original loop body of the codelet has been split in several smaller loops. The presence of several blocks induces branch instructions inside the codelets, and produces low performances. Such codelets are discarded before benchmarking. As an other example, MAQAO will detect spill code due to high register pressure. Spill code causes more memory transactions than expected, and usually decrease the performance of a codelet. Codelets with spill code are also discarded.

After the quality check with MAQAO, the remaining codelets are benchmarked. A program is generated, calling the codelets enough times to avoid timers problems

[112]. The codelets is also called a few times before the measurement to put the data in the correct memory configuration. Data structures are filled with random values, the characteristics of the constant performance codelet ensuring that input data have no impact on the codelet performance.

Along with the benchmarking of the codelets, the codelets for copying and/or prefetching the data are generated, and also benchmarked. Once the codelets have been tested, possibly in different input cache configuration, the original function has to be rebuilt, ensuring that the codelets are executed in the same condition.

2.2.7 Codelets (re-)composition

With the codelets evaluated, along with the possible memory transactions required, the function is recomposed. The first attempt to recomposition consists in combining the best optimized codelets for each original codelet extracted from the function. However, this recomposition may not be the best one, and may even be worse than the original function. Some overheads are hidden when evaluating the codelets independently, and they can consume all the improvement brought by the best codelets.

The first of these overheads is due to function calls. Extracted codelets have to be called from the recomposed function, which did not contained function calls originally. There is no way to evaluate the time spend in the functions calls before the recomposition of the function, and in the worst case, the overhead induced by these calls will be greater than the time gain brought by the codelet.

A second overhead is simply due to the addition of codelets for copy or prefetch. The time spend in unnecessary data movement will also consume a part of the overall speedup.

Another overhead can come from producing more possible codelets to optimize, before the extraction. A fission in a loop nest to generate new codelets eligible for the CPC framework can cause an array privatization. Data produced in the first part of the loop are not directly available for the secont part of the loop, and new data structures are created to send the data from one new codelet to the other. More memory transactions than in the original function will be issued, causing another reduction of the overall speedup.

This figure collects the speed-up obtained with different recomposition of codelets. From the second bar to the sixth bar, only one codelet is called from the tiled function, either the best or the worst for the selected loop, to show the difference existing between codelets from the same code. Bars labelled "All_worst" and "All_best" are for recomposition using respectively the worst codelets and the best ones, presented in previous bars. Using only the worst codelets brings a slowdown of 17%, while the best ones brings a speed-up of 9%. However, we observe that the "All_best" bar is not the best one. Despite its good stand-alone performance, the best codelet for Loop 2 consume more time when in the recomposition. The skewing applied on Loop 1 changes the data locality for Loop 2, causing the performance loss. As a fallback solution, we propose, when such cases occur, to try multiple recomposition, using the two or three best codelets (depending on the number codelet to keep the recomposition space small enough), and allowing also to leave the original code of the loop part being considered. Trying others recomposition, we see in Fig.2.2 that the function composed with the best codelets for the first and the fourth loop, and leaving the second loop intact gives the best prediction of 11%.

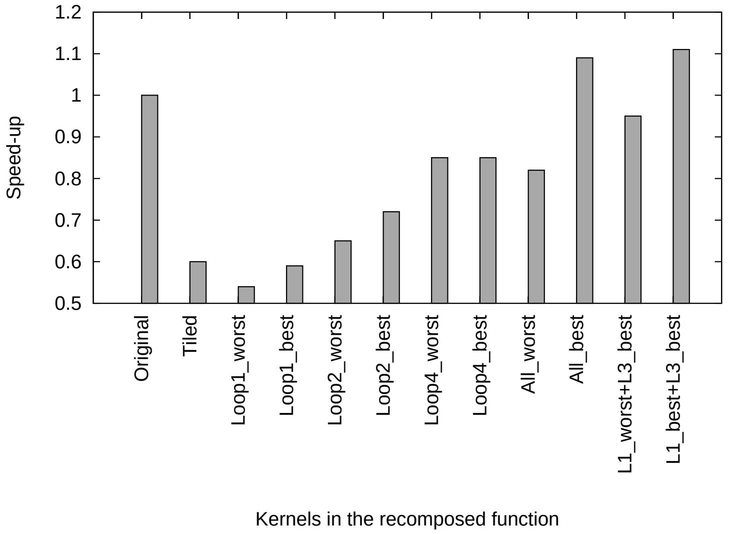


FIGURE 2.2 – Performance estimations for Gibbs depending on the combination of codelets involved.

2.3 Application to molecular simulation

Molecular simulation is an emerging technique which simulates a detailed system composed with several hundreds of molecules. With appropriate statistical formulae, one can, based on the simulation results, establish balance and transport properties, for comparison with experimental results. Molecular simulation can work on form of molecules, considering also several energy forms as repulsive-attractive energy (Lennard-Jones potential), electrostatic energy, polarization and torsion energy.

2.3.1 Presentation

GIBBS is a molecular simulation code developed by University of Orsay (Paris XI) and the French Oil Institute (IFP). GIBBS whole simulation aims to compute phase equilibria properties using a Monte-Carlo method on three-dimensional spaces (cube or parallelepiped). The simulation is driven by configuration files, and the results are written in output files. GIBBS offers the possibility to do multiple simulations in parallel for the same system, with variation of temperature, pressure and chemical potential. These simulations can exchange configurations following the *parallel tempering* method. However, only the sequential version was used to apply the *Constant Performance Codelet* framework. With the GIBBS code were provided 18 validation tests, representing several fluids with different pressure and temperature. The first test file was chosen to be our main benchmark.

The computation time for GIBBS is dominated by two codelet routines : *energy_ tot_ lj* and *energy_ mol_ lj* . These routines contribute respectively for 60% and 20% to the total execution time. A deeper analysis shows that these two functions call a third function, *energy_ lj*, which is inlined at compilation time. With the inlined function, *energy_ tot_ lj* presents a four-dimensional loop nest, whereas *energy_ mol_ lj* presents only a three-dimensional nested loop. Since more possibilities are offered for optimizations, like loop interchange, with the four nested loops, we focused our work on the *energy_tot_lj* function. The original nest loop is displayed in Fig.2.3.

```
for (i=start; i<end; i++)
  for (k=Start[i]; k<End[i]; k++)
    Tk = LJ[k]
    for (j=i+1; j<=end; j++)
      for (l=Start[j]; l<End[j]; l++)
        x = X[k] - X[l];
        y = Y[k] - Y[l];
        z = Z[k] - Z[l];
         x1 = x - b1 * round(x / b1);
         y1 = y - b2 * round(y / b2);
         z1 = z - b3 * round(z / b3);
        r2 = x1 * x1 + y1 * y1 + z1 * z1;
        if (r2 < mindist)
          energy = 2 * infinity;
        else
          energy=0;
          if (r2 < cutoff2)
            epsilon = Eps[Size * LJ[j] + tk];
            sigma = Sig2[Size * LJ[j] + tk];
            r2 = sigma / r2;
            r6 = r2 * r2 * r2;
            energy = epsilon * (rij6 - rij6 * rij6);
        Energy[j]= Energy[j] + energy;
        Energy[i]= Energy[i] + energy;
        sumEnergy = sumEnergy + energy;
```

FIGURE 2.3 – Initial Gibbs hot-spot nested loop.

2.3.2 Guided transformations for CPC framework

As we work on real life irregular codes, these codes do not originally fit the requirements describe in Section 2.2.1. Some initial transformations are required for the code to be a valid candidate to apply the CPC framework. These transformations are presented in the remainder of this section.

Tiling / Specialization

One of the main problem of the original code is loop bounds. For three of the four loops, bounds are not statically known, as also the size of the iteration space. Furthermore, two of the loops have bounds coming from array values, making the compiler unable to apply some optimizations, like vectorization on the innermost loop.

Tiling bypasses this kind of problems The iteration domain of a loop is partitioned in blocks with constant sizes. If the original domain is not a multiple of the chosen block size, a reminder loop is added after the tiled loop to look over the iterations not being swapped by it. With this transformation, the greater part of iterations execute code in statically bounded loops, allowing easy application of some optimizations, and the majority of branch instructions will be easier to predict.

However in this case, only the third loop (`for j`) will be tiled. Some profiling and dynamic analysis of the bound values revealed that the starting and the ending values of second and fourth loop were the same. Hence, these loops are executing only one iteration. In order to simplify the code, these two loops can be removed, and replaced by a statement fixing the value of iterators k and l to their beginning value, Start[i] and Start [j]. The third loop (`for j`) is the only remaining loop with its bound not statically known, and will be the only tiled loop.

Code fission (with array privatization)

The second problem preventing the application of the CPC framework is the presence of conditionals depending on input data in the body of the inner loop. Since the evaluation of constant performance codelets requires that conditionals depend only on loop counters, it is necessary to transform such code in straight line code removing conditionals, or to isolate the part of the code with the problematic feature.

In the general case, isolation is a good option. The isolation of a portion of code in a loop may induce some array privatization. Where a simple data produced in the loop was directly used in the isolated part, now the same data has to be stored in a conservative structure (like an array) until the first loop is finished and the loop containing the isolated part reaches the consumption of the data. The same behaviour will also happen for data produced in the isolated code and used in the remaining code after it.

Fig.2.4 presents the code of the loop nest after applying the preparation transformations. In this case, the fission was not necessary. Our test machine featuring an Itanium 2 processor, the instruction set of this architecture allow the transformation of `if...then...else...` constructs into a straight line block of code with conditional instructions. Both branches are executed, and predicate registers specify at runtime if the corresponding instruction should be executed, or replaced by a *nop* (no operation). The execution no longer depends on the value of the test, and the input data does not interfere anymore.

2.3.3 Codelet optimization

Now that the nested loop fits the criteria of a *Constant Performance Codelet*, they are extracted from the original code, in order to apply and evaluate the optimizations only on this small part. The optimization space can be very wide, and only transformations leading to some performance improvement will be presented and described in the following subsections. A large specter of generated codelets are displayed in Figures 2.6, 2.7, 2.8 and 2.9. Unrolled versions of these codelets have also been generated.

Fission and array privatization

Compilers apply optimizations considering only a restricted window of instructions. If the basic block the compiler tries to optimize is to large, it may drop useful transformations. With the MAQAO [36] tool, the quality of the assembly code has been observed. It appeared that the compiler failed to reorganize efficiently the code of the whole loop body, and the overall quality of the generated code was very poor.

In order to simplify the work of the compiler, fission transformations cut the loop body into several part, as shown in Fig.2.5.

In the large optimization space considered, different strategies are used concerning the loop fission. In the presented example, the loop is cut so that similar computations are gathered in the same codelet. Another fission version gathered instructions belonging to the same data dependence chain, but the obtained derived codelets are not as efficient as the presented version.

```
for (i=start; i<end; i+=STEPi)
  k = Start[i];
  Tk = LJ[k];
  for (j=i+1; j<=end; j+=STEP)
    for (jj=0; jj<STEP <= end; jj++)
      l = Start[j+jj];
      x = X[k] - X[l];
      y = Y[k] - Y[l];
      z = Z[k] - Z[l];
      x1 = x - b1 * round(x / b1);
      y1 = y - b2 * round(y / b2);
      z1 = z - b3 * round(z / b3);
      r2[jj] = x1 * x1 + y1 * y1 + z1 * z1;
    for (jj=0; jj<STEP && jj+j <= end; jj++)
      if (r2[jj] < mindist)
        energy = 2 * infinity;
      else
        energy=0;
        if (r2[jj] < cutoff2)
          epsilon = Eps[Size * LJ[j+jj] + tk];
          sigma = Sig2[Size * LJ[j+jj] + tk];
          r2 = sigma / r2[jj];
          r6 = r2 * r2 * r2;
          energy[jj] = epsilon * (rij6 - rij6 * rij6);
    for (jj=0; jj<STEP && jj+j <= end; jj++)
      Energy[j+jj]= Energy[j+jj] + energy[jj];
      Energy[i]= Energy[i] + energy[jj];
      sumEnergy = sumEnergy + energy[jj];
```

FIGURE 2.4 – Gibbs hot-spot function after preparation for CPC framework.

Skewing

An indirection in a computation can greatly degrade codelet performance. It prevents the compiler from applying aggressive optimizations, and the all the memory accesses also consume a lot of time.

Independent tests were performed on artificially made codes with some indirections. The purpose was to establish if eliminating the indirection from the body of the loop, and to move it in the call of the codelet, will allow the compiler to generate a better code, and to obtain better performance. Different type of distribution were used to fill the indirection array. Benchmarks were made with constant values ($Ind[i] = cst$), identical values ($Ind[i] = i$), strided values ($Ind[i] = k * i$ with k constant, typically 2 and 4) and random values ($Ind[i] = random(seed)$). For the tests with strided and random values, passing the indirection as an argument to the call of the codelet, and having it replaced with a scalar parameter in the codelet body, brought better performances. For the remaining cases, the performance was the same. Hence, trying to remove the indirection from the body of the first loop (extracted in Fig.2.6**(a)**) seemed the best way of improving the loop performance.

However, due to fission and multiple tiling, extracting the codelet in its current form, or after performing some loop interchange, is not sufficient to remove the indirection. Several loop counters are used in the index of the indirection array. In order to get rid of the indirection through extraction of the loop, skewing is necessary. Skewing consists in twisting the iteration space of nested loops, to swap the same amount of data in a different order. When skewing, new loops are created, replacing the previous ones, with new bounds defining the same iteration domain. In order to still be able to perform the extraction after applying the skewing, and

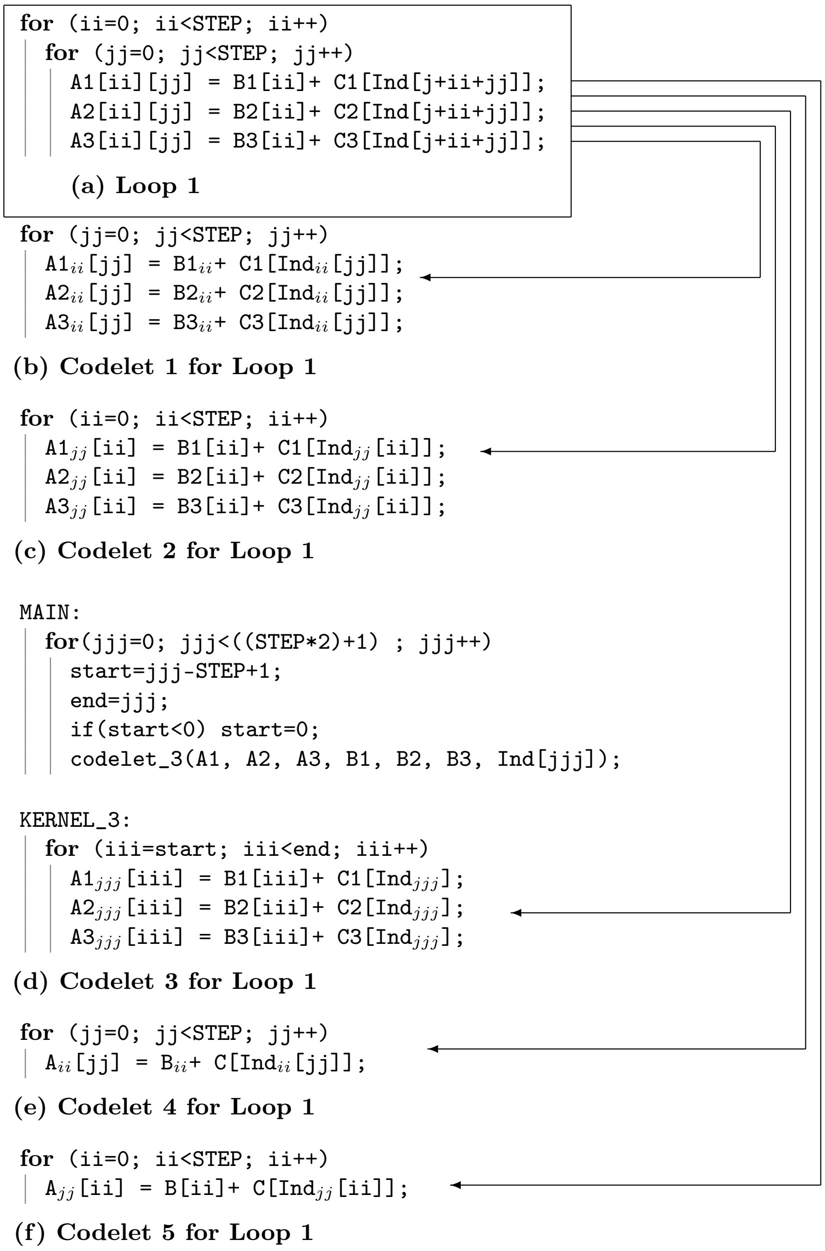
```
for (i=start; i<end; i+=STEPi)
  for (ii=0; ii < STEPi; ii++)
    k = Start[i+ii];
    Tk = LJ[k];
    xtemp[ii] = X[k];
    ytemp[ii] = Y[k];
    ztemp[ii] = Z[k];
  for (j=i+1; j<=end; j+=STEP)
    for (ii=0; ii < STEPi; ii++)
      for (jj=0; jj<STEP && jj+j+ii <= end; jj++)
        x[ii][jj] = xtemp[ii] - X[Start[j+ii+jj]];
        y[ii][jj] = ytemp[ii] - Y[Start[j+ii+jj]];
        z[ii][jj] = ztemp[ii] - Z[Start[j+ii+jj]];
    for (ii=0; ii < STEPi; ii++)
      for (jj=0; jj<STEP && jj+j+ii <= end; jj++)
        x1 = x[ii][jj] - b1 * round(x / b1);
        y1 = y[ii][jj] - b2 * round(y / b2);
        z1 = z[ii][jj] - b3 * round(z / b3);
        r2[ii][jj] = x1 * x1 + y1 * y1 + z1 * z1;
    for (ii=0; ii < STEPi; ii++)
      for (jj=0; jj<STEP && jj+j+ii <= end; jj++)
        if (r2[ii][jj] < mindist)
          energy = 2 * infinity;
        else
          energy=0;
          if (r2[ii][jj] < cutoff2)
            epsilon = Eps[Size * LJ[j] + tk];
            sigma = Sig2[Size * LJ[j] + tk];
            r2 = sigma / r2[ii][jj];
            r6 = r2 * r2 * r2;
            energy[ii][jj] = epsilon * (rij6 - rij6 * rij6);
    for (ii=0; ii < STEPi; ii++)
      for (jj=0; jj<STEP && jj+j+ii <= end; jj++)
        Energy[j+jj+ii]= Energy[j+jj+ii] + energy[ii][jj];
        Energy[i+ii]= Energy[i+ii] + energy[ii][jj];
        sumEnergy = sumEnergy + energy[ii][jj];
```

FIGURE 2.5 – Gibbs hot-spot function after some optimizations (fission and interchange).

not have the indirection in the loop body, two nested loops are be produced. The outermost loop swaps the indirection elements, without involvement of the second loop counter. When the innermost loop is extracted, the indirection becomes an argument in the codelet call, disappearing from the codelet body. The resulting skewed loop and codelet are displayed in Fig.2.6**(d)**.

Intrinsics specialization

The last optimization applied specifically to the GIBBS code is an unusual transformation. Source-to-source transformations, especially for nested loops, usually change the loop structure, the swapping order of elements, or simplify the code for the compiler with scalar promotion or array flattening. Here, no such thing is done. The main performance issue in the second loop comes from a call to a library function (namely function *round()*), as shown in the stand alone loop in Fig.2.7**(a)**. The function call is somehow unavoidable, and the compiler does not handle very well function calls, especially in the middle of a computation. The tricky part is



```
for (ii=0; ii<STEP; ii++)
  for (jj=0; jj<STEP; jj++)
    A1[ii][jj] = B1[ii]+ C1[Ind[j+ii+jj]];
    A2[ii][jj] = B2[ii]+ C2[Ind[j+ii+jj]];
    A3[ii][jj] = B3[ii]+ C3[Ind[j+ii+jj]];
```

(a) Loop 1

```
for (jj=0; jj<STEP; jj++)
  A1$_{ii}$[jj] = B1$_{ii}$+ C1[Ind$_{ii}$[jj]];
  A2$_{ii}$[jj] = B2$_{ii}$+ C2[Ind$_{ii}$[jj]];
  A3$_{ii}$[jj] = B3$_{ii}$+ C3[Ind$_{ii}$[jj]];
```

(b) Codelet 1 for Loop 1

```
for (ii=0; ii<STEP; ii++)
  A1$_{jj}$[ii] = B1[ii]+ C1[Ind$_{jj}$[ii]];
  A2$_{jj}$[ii] = B2[ii]+ C2[Ind$_{jj}$[ii]];
  A3$_{jj}$[ii] = B3[ii]+ C3[Ind$_{jj}$[ii]];
```

(c) Codelet 2 for Loop 1

```
MAIN:
  for(jjj=0; jjj<((STEP*2)+1) ; jjj++)
    start=jjj-STEP+1;
    end=jjj;
    if(start<0) start=0;
    codelet_3(A1, A2, A3, B1, B2, B3, Ind[jjj]);

KERNEL_3:
  for (iii=start; iii<end; iii++)
    A1$_{jjj}$[iii] = B1[iii]+ C1[Ind$_{jjj}$];
    A2$_{jjj}$[iii] = B2[iii]+ C2[Ind$_{jjj}$];
    A3$_{jjj}$[iii] = B3[iii]+ C3[Ind$_{jjj}$];
```

(d) Codelet 3 for Loop 1

```
for (jj=0; jj<STEP; jj++)
  A$_{ii}$[jj] = B$_{ii}$+ C[Ind$_{ii}$[jj]];
```

(e) Codelet 4 for Loop 1

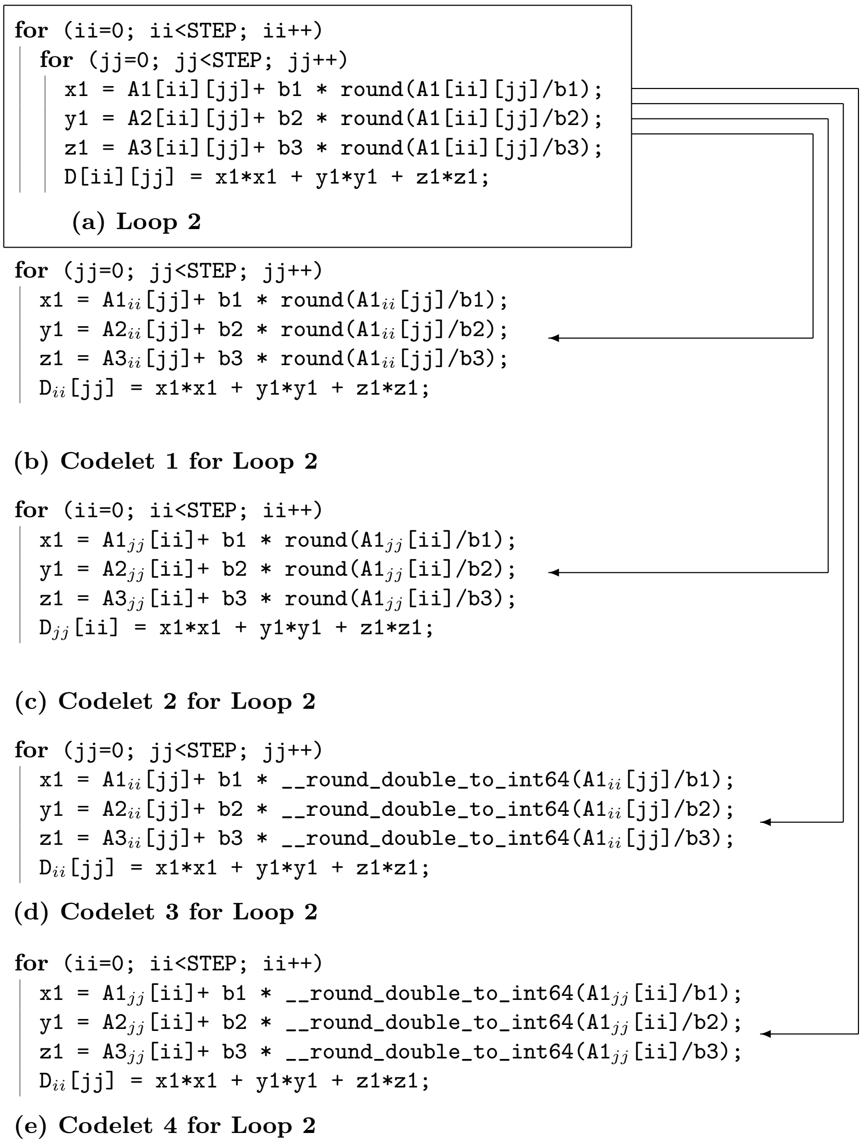
```
for (ii=0; ii<STEP; ii++)
  A$_{jj}$[ii] = B[ii]+ C[Ind$_{jj}$[ii]];
```

(f) Codelet 5 for Loop 1

FIGURE 2.6 – (a) Loop 1 out of context. (b), (c), (d), (e), (f) : computation codelets related to loop 1. (b) is obtained when the original innermost loop is extracted,(c) is obtained when the innermost loop is extracted after an interchange, (d) is obtained when a skewing is applied on the loops to remove the indirection from the codelet, (e) is obtained after a loop splitting separating each similar computation in independent codelets, (f) is obtained after an interchange and a loop splitting.

to know which function across all library or system calls available is the best to perform the task required.

An intrinsics function is a function available for a given language with an implementation handled specially by the compiler. It often directly refers to special assembly instructions for the target architecture. Therefore, a compiler dealing with



```
for (ii=0; ii<STEP; ii++)
  for (jj=0; jj<STEP; jj++)
    x1 = A1[ii][jj]+ b1 * round(A1[ii][jj]/b1);
    y1 = A2[ii][jj]+ b2 * round(A1[ii][jj]/b2);
    z1 = A3[ii][jj]+ b3 * round(A1[ii][jj]/b3);
    D[ii][jj] = x1*x1 + y1*y1 + z1*z1;
```

(a) Loop 2

```
for (jj=0; jj<STEP; jj++)
  x1 = A1_ii[jj]+ b1 * round(A1_ii[jj]/b1);
  y1 = A2_ii[jj]+ b2 * round(A1_ii[jj]/b2);
  z1 = A3_ii[jj]+ b3 * round(A1_ii[jj]/b3);
  D_ii[jj] = x1*x1 + y1*y1 + z1*z1;
```

(b) Codelet 1 for Loop 2

```
for (ii=0; ii<STEP; ii++)
  x1 = A1_jj[ii]+ b1 * round(A1_jj[ii]/b1);
  y1 = A2_jj[ii]+ b2 * round(A1_jj[ii]/b2);
  z1 = A3_jj[ii]+ b3 * round(A1_jj[ii]/b3);
  D_jj[ii] = x1*x1 + y1*y1 + z1*z1;
```

(c) Codelet 2 for Loop 2

```
for (jj=0; jj<STEP; jj++)
  x1 = A1_ii[jj]+ b1 * __round_double_to_int64(A1_ii[jj]/b1);
  y1 = A2_ii[jj]+ b2 * __round_double_to_int64(A1_ii[jj]/b2);
  z1 = A3_ii[jj]+ b3 * __round_double_to_int64(A1_ii[jj]/b3);
  D_ii[jj] = x1*x1 + y1*y1 + z1*z1;
```

(d) Codelet 3 for Loop 2

```
for (ii=0; ii<STEP; ii++)
  x1 = A1_jj[ii]+ b1 * __round_double_to_int64(A1_jj[ii]/b1);
  y1 = A2_jj[ii]+ b2 * __round_double_to_int64(A1_jj[ii]/b2);
  z1 = A3_jj[ii]+ b3 * __round_double_to_int64(A1_jj[ii]/b3);
  D_ii[jj] = x1*x1 + y1*y1 + z1*z1;
```

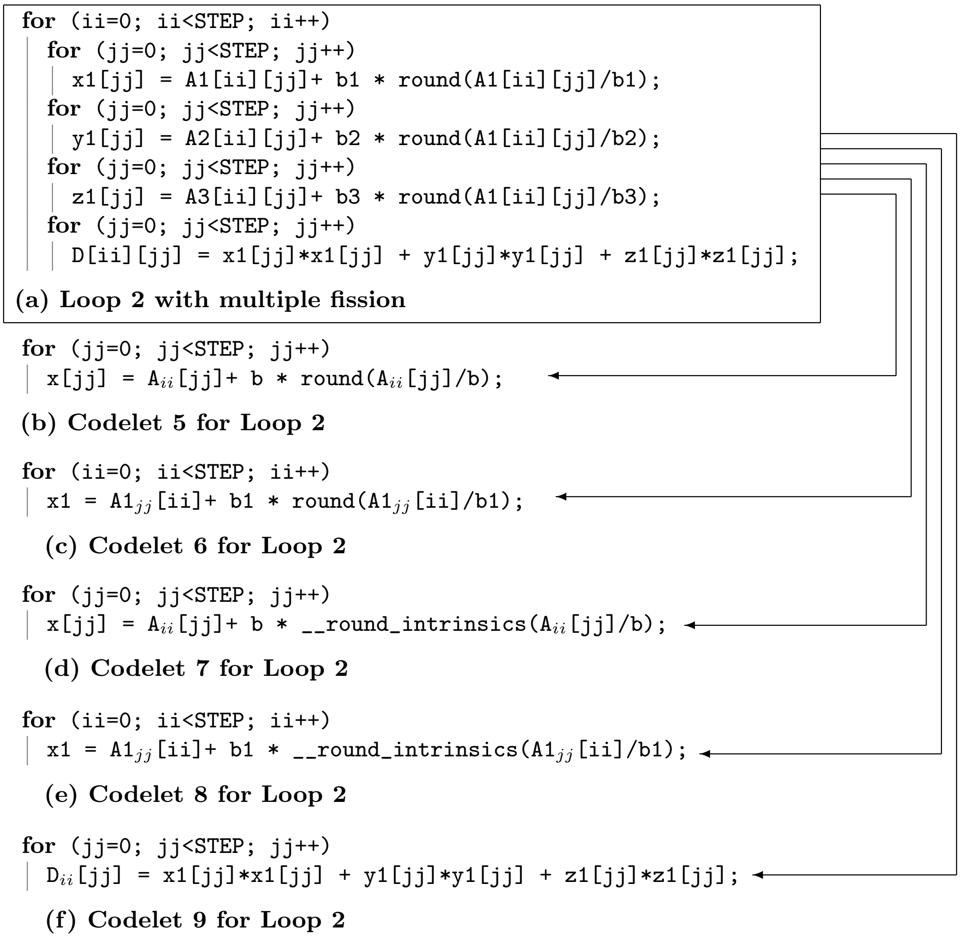
(e) Codelet 4 for Loop 2

FIGURE 2.7 – (a) Loop 2 out of context. (b), (c), (d), (e) : computation codelets related to loop 2. (b) is obtained when considering only the internal loop, (c) is obtained when an interchange is applied, (d) is obtained when replacement with intrinsics is applied on the original innermost loop, (e) is obtained when replacement with intrinsics is applied on the interchanged innermost loop.

an intrinsic function will most likely generates a much better executable code than when dealing with a library call sort of "unknown" to it.

An intrinsic function matches the *round()* function used in the second loop. Then, among the usual codelet optimizations, like interchange, versions with the intrinsic function *__round_double_to_int64()* were also tried. They returned the best improvement of performance for this loop. As explained before, this kind of transformation can be easily added to the spectrum of X-language to try it automatically.

This optimization was performed manually, as I knew the existence of the intrinsic function corresponding. However, one can easily imagine how to do it automa-



```
for (ii=0; ii<STEP; ii++)
    for (jj=0; jj<STEP; jj++)
        x1[jj] = A1[ii][jj]+ b1 * round(A1[ii][jj]/b1);
    for (jj=0; jj<STEP; jj++)
        y1[jj] = A2[ii][jj]+ b2 * round(A1[ii][jj]/b2);
    for (jj=0; jj<STEP; jj++)
        z1[jj] = A3[ii][jj]+ b3 * round(A1[ii][jj]/b3);
    for (jj=0; jj<STEP; jj++)
        D[ii][jj] = x1[jj]*x1[jj] + y1[jj]*y1[jj] + z1[jj]*z1[jj];
```

(a) Loop 2 with multiple fission

```
for (jj=0; jj<STEP; jj++)
    x[jj] = Aii[jj]+ b * round(Aii[jj]/b);
```

(b) Codelet 5 for Loop 2

```
for (ii=0; ii<STEP; ii++)
    x1 = A1jj[ii]+ b1 * round(A1jj[ii]/b1);
```

(c) Codelet 6 for Loop 2

```
for (jj=0; jj<STEP; jj++)
    x[jj] = Aii[jj]+ b * __round_intrinsics(Aii[jj]/b);
```

(d) Codelet 7 for Loop 2

```
for (ii=0; ii<STEP; ii++)
    x1 = A1jj[ii]+ b1 * __round_intrinsics(A1jj[ii]/b1);
```

(e) Codelet 8 for Loop 2

```
for (jj=0; jj<STEP; jj++)
    Dii[jj] = x1[jj]*x1[jj] + y1[jj]*y1[jj] + z1[jj]*z1[jj];
```

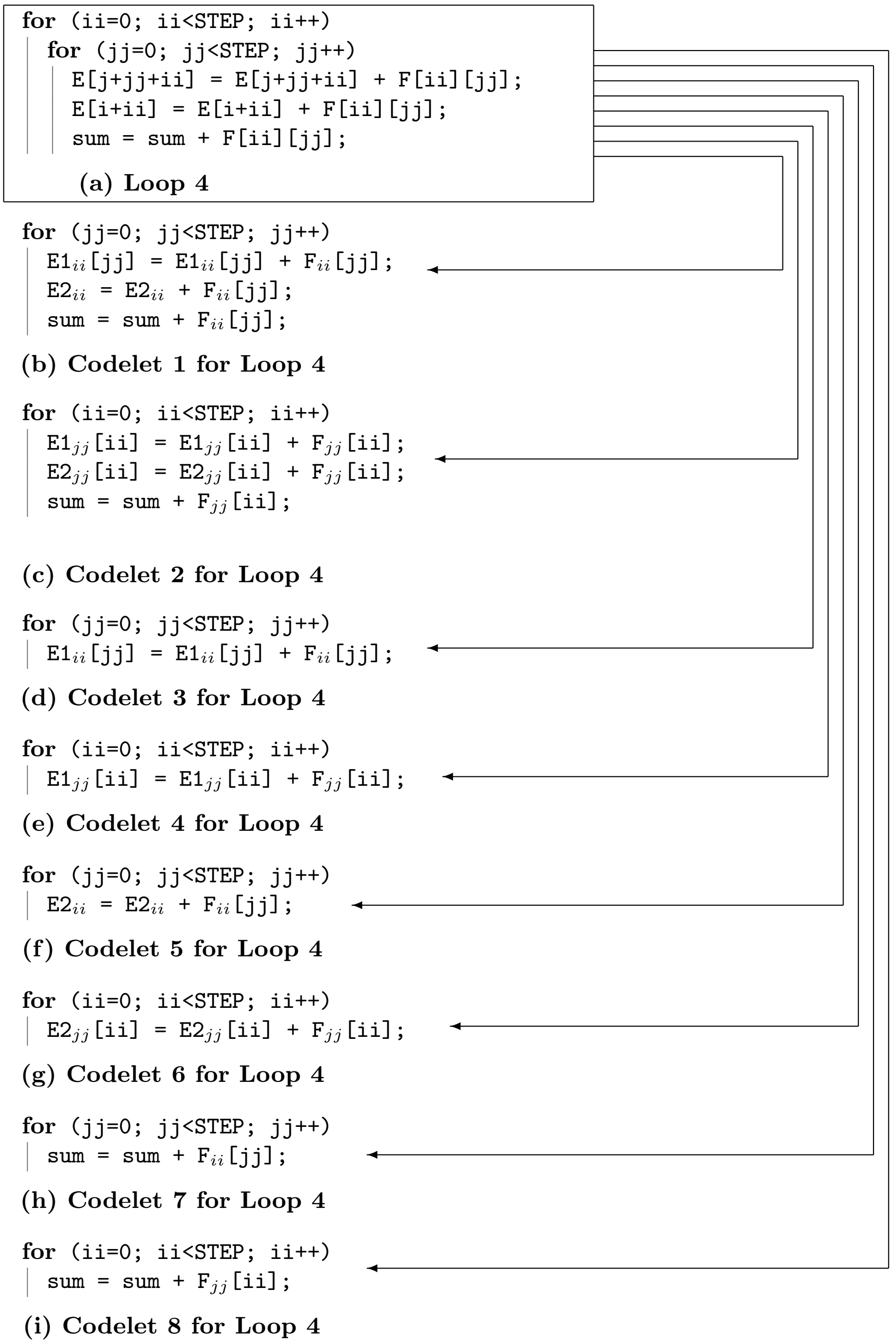
(f) Codelet 9 for Loop 2

FIGURE 2.8 – (a) Loop 2 out of context after fissions. (b), (c), (d), (e), (f) : computation codelets related to loop 2. (b) is obtained when considering only the first loop resulting of the fission, (c) is obtained when an interchange is applied on the first loop resulting of the fission, (d) is obtained when replacement with intrinsics is applied on the first loop, (e) is obtained when replacement with intrinsics is applied on the interchanged first loop, (f) is obtained when considering the last loop resulting of the fission.

tically. With a list of possibly used library functions matching their corresponding intrinsic functions available, a simple search when a function occurs inside a codelet will allow to replace it with a more efficient intrinsic function.

2.3.4 Performance evaluation

Once all the codelets are generated, they are evaluated with the method described in section 2.2.6. The best codelets for the GIBBS hot-spot nested loops are Codelet 3 for Loop 1 (presented in Fig.2.6**(d)**), Codelet 2 for Loop 2 (in Fig.2.7**(d)**), and Codelet 1 for loop 4 (in Fig.2.9**(b)**). They should be chosen to recompose the loop nest of function *energy_tot_lj*. However, as explained in section 2.2.7, the skewing done on the first best codelet does not interact well with the codelet for the second loop. As suggested, we tried several others recomposition, using the two best codelets for each loop, and allowing also to leave the original code for the loops. The best recomposition is predicted to be the one using the best codelets for Loop



```
for (ii=0; ii<STEP; ii++)
    for (jj=0; jj<STEP; jj++)
        E[j+jj+ii] = E[j+jj+ii] + F[ii][jj];
        E[i+ii] = E[i+ii] + F[ii][jj];
        sum = sum + F[ii][jj];
```

(a) Loop 4

```
for (jj=0; jj<STEP; jj++)
    E1_ii[jj] = E1_ii[jj] + F_ii[jj];
    E2_ii = E2_ii + F_ii[jj];
    sum = sum + F_ii[jj];
```

(b) Codelet 1 for Loop 4

```
for (ii=0; ii<STEP; ii++)
    E1_jj[ii] = E1_jj[ii] + F_jj[ii];
    E2_jj[ii] = E2_jj[ii] + F_jj[ii];
    sum = sum + F_jj[ii];
```

(c) Codelet 2 for Loop 4

```
for (jj=0; jj<STEP; jj++)
    E1_ii[jj] = E1_ii[jj] + F_ii[jj];
```

(d) Codelet 3 for Loop 4

```
for (ii=0; ii<STEP; ii++)
    E1_jj[ii] = E1_jj[ii] + F_jj[ii];
```

(e) Codelet 4 for Loop 4

```
for (jj=0; jj<STEP; jj++)
    E2_ii = E2_ii + F_ii[jj];
```

(f) Codelet 5 for Loop 4

```
for (ii=0; ii<STEP; ii++)
    E2_jj[ii] = E2_jj[ii] + F_jj[ii];
```

(g) Codelet 6 for Loop 4

```
for (jj=0; jj<STEP; jj++)
    sum = sum + F_ii[jj];
```

(h) Codelet 7 for Loop 4

```
for (ii=0; ii<STEP; ii++)
    sum = sum + F_jj[ii];
```

(i) Codelet 8 for Loop 4

FIGURE 2.9 – (a) Loop 4 out of context. (b), (c), (d), (e), (f), (g), (h), (i) : computation codelets related to loop 4. (b) is obtained when the original innermost loop is extracted,(c) is obtained when the innermost loop is extracted after an interchange, (d), (f), (h) are obtained when considering each inner loops resulting of a fission, (e), (g), (i) are obtained after an interchange is applied on each of these loops.

1 and Loop 4, with the original code for Loop 2.

The performance of this recomposition is displayed in the first cluster of bars in Fig.2.24. One can recognize in the first two bars the same speed-up already presented in Fig.2.2, the first one being the original code, and the second the predicted 11% speed-up with the best recomposition. A recomposition being done, the new function is executed to measure the actual performance. A 9% speed-up is obtained, represented with the last grey bar. To be thorough, we executed several others recomposition among the best ones to ensure that the one presented in Fig.2.24 is indeed the best possible recomposed function.

For this first test, 78 different optimized codelet versions have been generated. Among all these codelets, 30 were discarded during the selection phase with MAQAO, remaining 48 codelets to evaluate. Tens of thousands of recomposition are possible with the remaining codelets, but only a handful of these codelets are considered for recomposition, exhibiting good enough performance. At last, only 7 different compositions have been tried to find the best version of the function (which is the third composition we tested).

On the GIBBS simulation, we managed to find the best speed-up possible executing the whole code only two times, and benchmarking only small part of the hot-spot function, with very different optimization tried. The actual measured performance is similar to the predicted one, ensuring that these small benchmarks are sufficient to predict the behaviour of the complete code, if selected in the hot-spot of the program.

2.4 Application to genomic code

The BLAST family (for Basic Local Alignment Search Tool [6]) is a collection of programs widely used for searching protein and DNA databases to find similarities between two sequences. The most known use of BLAST programs is to compare newly sequenced genomes with protein data banks in order to discover where genes are located. The family is composed of several variations : programs performing protein/protein comparison, nucleotide/nucleotide comparison, protein/translated nucleotide, and so on... The variety `TBLASTN` specifically search a **t**ranslated **n**ucleotide database, using a protein query.

2.4.1 Presentation

The index-TBLASTN algorithm [74] is a first attempt to parallelize BLAST-like programs on parallel platforms. Contrary to TBLASTN, this version does not aim to search large database. Instead, it focuses on comparing consequent amount of data, that is a complete genome and a protein bank. The main difference is that instead of indexing the query (sequences from the protein bank), index-TBLASTN indexes the complete genome following the 6 reading frame and performs a dynamic indexing of the protein bank. Each element in the index is enhanced with neighborhood information to avoid costly random memory access.

Profiling the code, it appears that the most time consuming procedure is the *ComputeDistance* function with the hot-spot loop nest displayed in Fig.2.10.

This function takes as input two index structures : one from the genome (Qry) and the other from the protein bank ($FULL\_INDEX$). Both indexes are related to the same seed. If N is the number of elements in the genome index, and M the number of elements in the protein bank index, then the procedures computes NxM scores based on an amino acid substitution matrix. The score computation is highlighted with red color in Fig.2.10.

```
for (j=0; j<rq->Qry->nb; j++)
  for (i=rq->offset; i<rq->offset+rq->nbelt; i++)
    score = 0;
    maxi = 0;
    for (k=0; k<SIZE_BLOCK; k++)
      score = score + MATRIX[A[rq->Qry->seq[j][k]]] [A[INDEX[i].seq[k]]];
      if (score<0) score = 0;
      else if (maxi<score) maxi = score;
    if (maxi > X1)
      rq->Res[r].idx_nt = FULL_INDEX[i].idx;
      rq->Res[r].idx_prot = rq->Qry->idx[j];
      rq->Res[r].num_seq_prot = rq->Qry->num_seq[j];
      r++;
```

FIGURE 2.10 – Original hot-spot in BLAST function.

2.4.2 Transformations for CPC framework

As with GIBBS implementation, the loop nest is not yet suited to apply the CPC framework. The bounds of two loops are dependent of input data. The iteration space is not known at compile time, which can prevent the compiler, and the programmer, to apply some loop transformations. In addition to the loop bounds, most instructions in the loop body are under the influence of conditionals, depending again on the input data.

Tiling and fission

We encountered the same limitations with the GIBBS program, and they will be dismissed with the same transformations than the ones used in Section 2.3.2.

- A tiling is used to separate the iteration space in blocks with constant sizes. Again, a reminder loop is added to swap iterations not fitting in these blocks.
- Then, fission is applied to isolate conditional blocks from the elaborate memory access using several indirections. Our Itanium 2 machine is not the only target for this code, and it will also run on processors without the possibility to transform `if...then...else...` blocks into predicated straight lines of code. The fission is necessary for the CPC framework to directly use the transformed function on those machines.

Now, the computation loop presents all required characteristics to be a constant performance codelet. The framework can be used to optimize this loop nest.

2.4.3 Codelet optimization

The computation presents features which can greatly degrade the performance. The access to the matrix is done with no less than four indirections. A first step should be to compute as soon as possible parts of the indirection, to have data, or at least addresses, available when they are needed. To realize all the memory accesses when the data is needed will cause time loss due to the memory latency. The program will stall the computation, waiting for the required data to be loaded. However, extracting the codelet already perfom all scalar promotions and array flattening possible. To actually compute the indirections in advance inside the codelet will be redundant with the simplification embedded in the codelet extraction. This

```
for (J=0; J<rq->Qry->nb; J+=SIZEJ)
  for (I=rq->offset; I<rq->offset+rq->nbelt; I+=SIZEI)
    n=0;
    for (j=J; j<J+SIZEJ ; j++)
      for (i=I; i<I+SIZEI ; i++)
        for (k=0; k<SIZE_BLOCK; k++)
          t[n][k]=(char)MATRIX[A[rq->Qry->seq[j][k]]] [A[INDEX[i].seq[k]]];
        indexi[n]=i;indexj[n]=j;
        n++;
    N=n;
    for (n=0; n<N; n++)
      score = 0;
      maxi = 0;
      for (k=0; k<SIZE_BLOCK; k++)
        score = score + t[n][k];
        if (score<0) score = 0;
        else if (maxi<score) maxi = score;
      if (maxi > X1)
        rq->Res[r].idx_nt = FULL_INDEX[indexi[n]].idx;
        rq->Res[r].idx_prot = rq->Qry->idx[indexj[n]];
        rq->Res[r].num_seq_prot = rq->Qry->num_seq[indexj[n]];
        r++;
```

Figure 2.11 – Distributed version of hot-spot loop nest in BLAST.

redundancy of computations and memory accesses may induce worse performance instead of a speed-up.

Code motion and array privatization

In this code, the memory access of computed data causes two issues. The first one, described before, is the number of indirections for one data access. The other one is the proximity between the memory request and the computation using this data, the computation being issued right after the memory request. Until the data is fetched, the program will stall, and a stall will occur for each computation.

A code motion, with a fission, is applied to separate the heavy memory access from the computation requiring the data. Since they end up in different codelet, and array is created to pass data from a codelet to another. This succession of transformations realized a memory copy from a data structure with an intricate acces pattern, to a data structure with a classical pattern access. The computation remains close to a memory access, but the simplicity of this access decrease the impact of stalls due to memory latency. The resulting code is displayed in Fig.2.11.

Unroll

If the compiler applies optimization considering only a restrictive frame of instructions, it also needs to have a certain amount of instructions to play with. If a loop body is too simple, the compiler can apply only a limited set of optimizations. The compiler will not be able, for example, to interleave efficiently memory accesses and computations. In our code example, the computation loop contains only one line code. The complexity of the memory access will produce several instructions, heavily connected. The compiler may not be able to produce an efficient executable, since it is not possible to reorganise these instructions. To provide more possibility of optimizations, several unroll factor were tried on the copy codelets. The best result is obtained with the code in Fig.2.12, with an unroll factor of 4 on the outer loop (for j), and on the middle loop (for i).

```
 for (j=J; j<J+SIZEJ ; j+=4)
    for (i=I; i<I+SIZEI ; i+=4)
     for (k=0; k<SIZE_BLOCK; k++)
       ACOPY_4x4(i,j,n);
     n+=16;
    for (; i<I+SIZEI && i<rq->offset+rq->nbelt; i++)
     for (k=0; k<SIZE_BLOCK; k++)
       ACOPY_1x4(i,j,n);
     n+=4;
⋮
for (n=0; n<N; n++)
  score = 0;
  maxi = 0;
  for (k=0; k<SIZE_BLOCK; k++)
    score = score + t[n][k];
⋮
```

Figure 2.12 – Unrolled version of hot-spot loop nest in BLAST.

To lighten the presentation of the transformed codes in this dissertation, subroutine calls are used. It will ease the reading of presented codelets, since a computation codelet having sustained a factor 4 unroll on loops (for i) and (for j) will take only one line through the subroutine code, instead of sixteen computation lines being really implemented. Some used subroutines are detailed to clarify the actual code hidden behind their calls. Subroutine copy calls are described in Fig.2.13. Subroutine copy calls for the vectorized version are described in Fig. 2.15, and computation calls for the same vectorized version are described in Fig.2.16.

Since the unroll factor may not be a natural divider of the tile size, a reminder is added after the unrolled loop to compute sequentially the solitary elements.

Vectorization

On most architecture, vectorization is a key transformation to obtain good performance. However, indirections prevent useful vectorization : since the considered data are not contiguous, when doing a vectorized computation, one has no way to know which elements are computed along with the current one. space.

Even considering that elements form a compact block in memory, they are not swapped sequentially. The computed elements along with the current one in the vector cannot be known. All elements need to be done in the query order, to be sure that no elements were missed. Doing so, the same computation will occur several times : when the considered element is the current one, and each time the element is part of another element vector.

To apply safely and efficiently the vectorization, vectors are build with consecutive elements on the computation, and not consecutive elements in memory. This way, the vectorized computation will really consider several consecutive iterations of the original computation.

```
ACOPY(I,J,N)
  t[(N)][k]=(char)MATRIX[A[rq->Qry->seq[(J)][k]]] [A[INDEX[(I)].seq[k]]];

ACOPY_4x1(I,J,N)
  ACOPY(I,J,N);
  ACOPY(I+1,J,N+1);
  ACOPY(I+2,J,N+2);
  ACOPY(I+3,J,N+3);

ACOPY_4x4(I,J,N)
  ACOPY_4x1(I,J,N);
  ACOPY_4x1(I,J+1,N+4);
  ACOPY_4x1(I,J+2,N+8);
  ACOPY_4x1(I,J+3,N+12);
```

Figure 2.13 – Subroutine calls used to represent copy instructions in a codelet. In this example, loops (for j) and (for i) in the codelet have been unrolled with a factor of 4. Hence, **ACOPY_4x4** subroutine represent sixteen copy statement in the actual implementation.

```
⋮
      for (n=0; n<NV-7; n+=8)
        for (k=0; k<SIZE_BLOCK; k++)
          VCOMPUTE_8(vt,k);
      for (; n<NV; n++)
        for (k=0; k<SIZE_BLOCK; k++)
          VCOMPUTE(vt,0,k);
      for (n=0; n<N-7; n+=8)
        for (k=0; k<SIZE_BLOCK; k++)
          COMPUTE_8(k);
      for (; n<N; n++)
        for (k=0; k<SIZE_BLOCK; k++)
          COMPUTE(0,k);
⋮
```

Figure 2.14 – Vectorized version of hot-spot loop nest in BLAST.

Once the vectors are ready, the score is computed. In this case, the compute code is rewritten with intrinsics functions, which, among others, allow to find the maximum value for each element between two vectors. Hence, the conditional test **if**(score<0) score = 0; (resp. **elseif**(maxi<score) maxi = score;) is replaced with vector intrinsic _mm_max_pu8(vscore N, zero); (resp. _mm_max_pu8(vscore N, vmaxi;), as shown in Fig.2.16 first subroutine representation.

Along with the vectorization, a new unroll is applied. Vector registers using 8 elements, an unroll factor of 8 on the elements now consume as many registers as the original sequential computation. Plenty of vector registers remain to unroll the outer loop. Different unroll factor is tried, and all versions of codelets (unrolled and vectorized, vectorized, unrolled, and initial) are evaluated to find the best value.

```
VCOPY(I,J,N)
  c_N=MATRIX[A[rq->Qry->seq[(J)][k]]] [A[INDEX[(I)].seq[k]]];

VCOPY_1x8(I,J,N)    COPY(I,J,0);
    COPY(I,J+1,1);
    COPY(I,J+2,2);
    COPY(I,J+3,3);
    COPY(I,J+4,4);
    COPY(I,J+5,5);
    COPY(I,J+6,6);
    COPY(I,J+7,7);
    vt[(N)][k] = _mm_set_pi8((char)c7,(char)c6,(char)c5,(char)c4,(char)c3,(char)c2,(char)c1,
```

Figure 2.15 – Subroutine calls used to represent copy instructions in a codelet. In this example, loops (for j) in the codelet has been unrolled with a factor of 8. The new data structures is filled to prepare the vectorization applied on computation codelets.

```
VCOMPUTE(V,N,K)
  vscore_N= _mm_add_pi8(vscore_N,(V)[n+(N)][(K)]);
  vscore_N= _mm_max_pu8(vscore_N, zero);
  vmaxi_N= _mm_max_pu8(vscore_N, vmaxi_N);

VCOMPUTE_4(V,K)
  VCOMPUTE(V,0,K)
  VCOMPUTE(V,1,K)
  VCOMPUTE(V,2,K)
  VCOMPUTE(V,3,K)

COMPUTE(N,K)
  score_N = score_N + t[n+(N)][K];
  if (score_N<0) score_N= 0;
  else if (maxi_N < score_N) maxi_N= score_N;

COMPUTE_4(K)
  COMPUTE(0,K)
  COMPUTE(1,K)
  COMPUTE(2,K)
  COMPUTE(3,K)
⋮
```

Figure 2.16 – Subroutine calls used to represent computations in a codelet. **VCOMPUTE** stands for codelets having vectorized computation, while **COMPUTE** stands for original sequantial computations. Loop (for n) has been unrolled four times in each codelet.

An unroll factor of 8 on the outer loop of the vectorized version brings the greater speed-up.

2.4.4 Performance evaluation

For this program, 28 different optimized codelet versions have been generated. None of these codelets were discarded due to the quality of their code. With these codelets, only forty different recombinations are possible. At last, only 16 different compositions have been tried to find the best version of the function. The code using the new recomposed function is executed with two input files : TestProt1 and TestProt2. The two files test two different proteins, with the second one being larger than the first one. A prediction is established beforehand for each input files. The predictions are forecasting a speed-up of a factor x1.96 for the second test and up to a factor x2.32 for the first input. Performance figures are presented in the two last cluster of bars in Fig.2.24. We can see that if the actual measured speed-up is within a 10% error margin of the prediction for TestProt1, achieving a factor of x2.23, the performance for the second test is less impressive than the prediction. Still achieving a speed-up of 66%, there is a gap between predicted performance and measured one.

This difference is due to the size of TestProt2 tests. When the codelets are benchmarked, data is assumed to be in cache. But for this test, all data does not fit in cache, and useful data is flushed between iteration of the outer loop. These data must be fetched from memory, taking more time than expected. Our memory model being very simple, and the data of TestProt2 benchmark just a little larger than the cache size, we have two possible ways to evaluate the codelet : with data in memory or with data in the last cache level. If we assume that the data is in memory, the prediction is greatly underestimated. Most of the data stay in cache in the computation, and the gain from cached memory accesses will not be evaluated. Assuming that data is in the last cache level, we obtain a closer, but overestimated, prediction. Some data are fetched from memory in the recomposed function, and the latency decreases the predicted speedup.

However, once again, we see that if the data are in the assumed memory configuration, as for TestProt1 input file, we can determine the overall performance of an optimized code by evaluating only codelets from the hot-spot function of the program.

2.5 Application to quantum chromodynamics simulation

Quantum chromodynamic (QCD) is the theory of strong interaction in the domain of subnuclear physics. Lattice QCD (LQCD) is a numerical method based on QCD's first principles, the only one able to compute reliably many quantities of high scientific relevance. It is based on a discretisation of space time and a Monte-Carlo method. The system being an extremely complex one and the number of degrees of freedom being of the order of a billion today, a number promised to increase in the future, LQCD needs heavy, efficient and cheap enough computing tools (hardware and software).

QCD is a hot subject, which produces two consecutive ANR projects, PARA and PetaQCD, involving the University of Versailles. Focus on software and hardware parts produces a great number of deliverables during these projects.

The goal of the calculation is to produce, according to a given probability law given by the theory, a wide statistical sample of "gauge configurations," each of which being a large file of complex numbers. The generally used algorithm is called "hybrid Monte-carlo" and it combines two steps. The first one is the calculation of a "trajectory," according to a Hamiltonian in a complex dynamical space, which leads from one gauge configuration to the next one of the sample, in such a way that the probability is approximately conserved. The second step performs a Monte-Carlo test to enforce the wanted probability law. The time consuming part is the computation of the trajectory. It is mainly linear algebra. It manipulates repeatedly a very large sparse matrix named “Dirac operator.”

Assessing the performance and efficiency on new architectures and based on different algorithmic representation of this problem is important to approach the computational power needed for this problem. This computation tends to have low utilization and efficiency on most general-purpose computing facility leading to inefficient power consumption and unrealistic demands on the number of needed computational nodes. Building special machine for simulating the Lattice QCD problem has been a widely used practice [17, 18, 14]. The motive for building specialized computing facilities with all the associated overheads is the enormous computational power needed in addition to the special characteristics of the computation of the Lattice QCD.

For large lattices the space of gauge configurations is a variety with dimensionality of the order of tenths of billions. Only a Monte-Carlo method allows such a huge calculation. To estimate the average values of the physical quantities we need representative samples of gauge configurations (say about 5000) for every set of parameters, generated according to the above-mentioned probability law. The Hybrid Mont Carlo (HMC) algorithm [38]is used to generate these samples. In the following discussion, we will consider an HMC implementation achieved by the ETMC collaboration [102, 101].

2.5.1 Presentation

The computation time for Lattice QCD is dominated by few kernel routines. The main kernel routine, called Hopping Matrix, is contributing about 90% of the total execution time [107]. This routine is the major one for computing the actions of Wilson-Dirac operator. As outlined in Equation 2.1 below, the actions of Dirac operator involve a sum over quark “spinors”($\psi(i)$) multiplied by a gluon gauge link ($U_\mu(i)$) through the spin projector.

$$\chi(i) = \sum_{\mu=x,y,z,t} \kappa_\mu \left\{ U_\mu(i)\,(I-\gamma_\mu)\,\psi(i+\hat{\mu}) + U_\mu^\dagger(i-\hat{\mu})\,(I+\gamma_\mu)\,\psi(i-\hat{\mu}) \right\} \tag{2.1}$$

In Hopping_Matrix, the four-dimensional space-time continuum is simulated by a four-dimensional lattice, with quark quantum fields, represented by spinors, on each lattice site and gluon quantum fields, represented by SU(3) matrices, on each link between these sites. The calculation aims at computing the average values of physical quantities, which are functions of these fields, according to a probability distribution also depending on the fields, and derived by a discretisation procedure from the basic QCD Lagrangian. The gauge field, defined on links, are SU(3) matrices going from a site into the four positive directions of the space-time. SU(3) refers to matrices with three colors of quarks that are unitary and of unit determinant. The spinors are represented by four SU(3) vectors, each composed of three complex variables.

As a result, the heart of the simulation code is a loop (or a loop nest) swapping a four-dimensional lattice to update each site like a 4D Jacobi stencil (see Fig.2.17), requiring the contribution of direct neighbors in each dimension. Each of the eights needed sites being a spinor, and each coefficient being a SU(3) matrices, the complexity of the overall lattice update explains that it is the most consuming function in the program.

When implemented, the four-dimensional space has been flattened to erase the cost a four nested loops, and only one loop is used to swap the whole volume. As

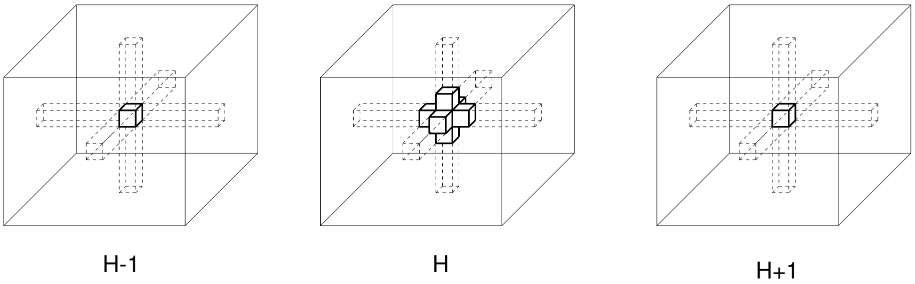


Figure 2.17 – Jacobi stencil pattern displayed on three consecutive hyperplanes of a 4D space. The update of the central node in hyperplane **H** requires contributions from its direct neighbours in each dimension.

a first step towards a better reuse of data between site updates, the computation over the volume is split in two distinct parts : a first loop for computing only even sites of the lattice, a second loop for the odd sites. A data dependence analysis of the loops shows that each direction if each loop is composed with two independent data stream. The loop structure is displayed in Fig. 2.18, with the contributions of each direct neighbors.

```
for(i=0; i<(VOLUME)/2; i++)
    computations_Even_Direction+0();
    computations_Even_Direction-0();
    computations_Even_Direction+1();
    computations_Even_Direction-1();
    computations_Even_Direction+2();
    computations_Even_Direction-2();
    computations_Even_Direction+3();
    computations_Even_Direction-3();

for(i=0; i<VOLUME/2; i++)
    computations_Odd_Direction+0();
    computations_Odd_Direction-0();
    computations_Odd_Direction+1();
    computations_Odd_Direction-1();
    computations_Odd_Direction+2();
    computations_Odd_Direction-2();
    computations_Odd_Direction+3();
    computations_Odd_Direction-3();
```

Figure 2.18 – Original code structure of function Hopping_Matrix. Computation on even and odd points of the lattice are split in two separate loops. Computation of the neighbour contributions in each direction is composed of two data stream, independent from one another.

On the difficulty of applying complex optimization

The code we were presented was composed of several identical functions. An original general purpose function written in C language, and other functions specialized for x86 and bluegene architectures (using intrinsics functions). All these functions were duplicated to have a sequential and a parallel version of each. Our role was to study the behaviour of the general purpose code on an Itanium 2 powered machine, and to optimize it for this architecture.

```
_vector_add(r,s1,s2) \
    (r).c0.re=(s1).c0.re+(s2).c0.re; \
    (r).c0.im=(s1).c0.im+(s2).c0.im; \
    (r).c1.re=(s1).c1.re+(s2).c1.re; \
    (r).c1.im=(s1).c1.im+(s2).c1.im; \
    (r).c2.re=(s1).c2.re+(s2).c2.re; \
    (r).c2.im=(s1).c2.im+(s2).c2.im;
```

FIGURE 2.19 – Actual code inlined when calling "_vector_add" subroutine.

The considered function is written in C language, but real line codes are not visible. With the computation applying on complex numbers, included in specific structures (like SU(3) matrices or vectors), each simple operation between these structures requires several code lines. To relax the work of the original programmer, each complex operations is an inlined subroutine. Only subroutine calls are visible in the loop body, as shown in Fig2.20.

```
for(i = 0; i < (VOLUME)/2; i++)
  _vector_add(psi, rs.s0, rs.s2);
  _su3_multiply(chi,(*iU),psi);
  _complex_times_vector((*phi[ix]).s0, ka0, chi);

  _vector_add(psi, rs.s1, rs.s3);
  _su3_multiply(chi,(*iU),psi);
  _complex_times_vector((*phi[ix]).s1, ka0, chi);
```

FIGURE 2.20 – Macro calls for direction +0 in function Hopping_Matrix.

If the underlying computation, i.e. the succession of operations, is clearly visible, the code structure prevented the possibility to gather part of different operations, in order to benefit from memory locality or register usage. Since the loop body is very large (each loop contains over 30 macros, composed with at least 6 computational code lines, like the one in Fig. 2.19), the code was not rewritten and the work was done on this version of the function.

2.5.2 Codelet Optimization

The loop is a good candidate for the CPC framework. The conditions described in section 2.2.1 are fulfilled. The loop bounds are statically known at compile time; control flow and data flow are independent of the input files.

Loop Fission

As explained before in Section 2.3.3, if the code in the loop is too large, the compiler will not be able to apply as many and aggressive optimizations as it could. In this function, 1204 floating-point operations are nearly equally divided between the two loops (the second one being a little larger than the first on). Considering also the memory transactions required to load data for these 1204 operations, a loop body is too large to be efficiently handle by the compiler. In order to reduce the code size, one will want to apply first some fission on the loop. Here, applying fission to isolate each computation on a direction is natural. Data produced by these computations are already stored in an array, thus avoiding the cost of an allocation of a new data structure to store the temporary results. Isolating each directions will not cause to increase the memory usage. The structure of the resulting code is presented in Fig. 2.21.

```
for(i=0; i<(VOLUME)/2; i++)
| computations_Even_Direction+0();
for(i=0; i<(VOLUME)/2; i++)
| computations_Even_Direction-0();
for(i=0; i<(VOLUME)/2; i++)
| computations_Even_Direction+1();
for(i=0; i<(VOLUME)/2; i++)
| computations_Even_Direction-1();
⋮
```

Figure 2.21 – Tiled code structure of function Hopping_Matrix.

With all directions isolated, we end up with several smaller codelets to consider.

Tiling

To increase data reuse, or retrieve the one lost with the fission, a tiling is applied on each loop. Then the outer loop on (VOLUME) is fused, in order to produce the code structure displayed in Fig. 2.22.

```
for(i=0; i<(VOLUME)/2; i+=TILE)
| for(j=0; j<TILE; j++)
| | computations_Even_Direction+0();
| for(j=0; j<TILE; j++)
| | computations_Even_Direction-0();
| for(j=0; j<TILE; j++)
| | computations_Even_Direction+1();
| for(j=0; j<TILE; j++)
| | computations_Even_Direction-1();
⋮
```

Figure 2.22 – Tiled and fissioned code structure of function Hopping_Matrix.

With a tiling value of 128 iterations, we manage to keep all possible data reuse in the closest cache memory available (L2 for floating-point data on Itanium 2). If computations in different directions use the same structures, a small enough tiling will allow the data to remain in memory between these computation. With an indirection giving the next element to compute, data reuse between consecutive iterations of a loop is highly improbable. However, it is possible that iterations using consecutive data are not very far one from another. Then, a greater tile value than the distance of these two iterations may allow the common elements to remain in close memory. With the right value, data reuse between loop iterations and computations in different direction is maximized.

Code motion and Fission

After the first fission to separate each direction computation in a different codelet, too many operations remain in each codelets, with an average of 75 floating-point operations per codelets. Additional fissions will be applied to diminish the number of computations per codelets. As shown in Fig.2.18, operations composing the computation of each direction are divided among two independent data flows. For each codelet containing a direction computation, a code motion will be applied to gather all operations belonging to the same data flow. Then a fission will cut the loop, separating the two independent data flows in new codelets.

A data dependence analysis of the code show that each codelet (i.e. each direction) is composed with two independent data flows. Hence, with some code motion, it is possible to isolate the streams. A new fission then can split the codelet in two new ones, with only half the computation in it. The resulting code structure for the loop is displayed in Fig. 2.23.

```
for(i=0; i<(VOLUME)/2; i+=TILE)
    for(j=0; j<TILE; j++)
        computations_Even_Direction+0_ids1();
    for(j=0; j<TILE; j++)
        computations_Even_Direction+0_ids2();
    for(j=0; j<TILE; j++)
        computations_Even_Direction-0_ids1();
    for(j=0; j<TILE; j++)
        computations_Even_Direction-0_ids2();

⋮
```

FIGURE 2.23 – Reorganized code structure of function Hopping_Matrix.

Some other fissions will be applied on these new codelets to separate the subroutines of each computation, creating codelets with a more manageable set of operations. However, these computations are part of the same data stream. Where a temporary value was enough to pass the result of an operation to the next computation, now an array is required to pass all the results of the same operation to the next codelet, realizing the next computation. The array privatization produces more memory transactions, and the generated codelets were not as efficient as the ones containing the computation of a whole data flow.

2.5.3 Performance evaluation

For this first test, 78 different optimized codelet versions have been generated. Among all these codelets, 30 were discarded during the selection phase with MAQAO, remaining 48 codelets to evaluate. Tens of thousands of recomposition are possible with the remaining codelets, but only a handful of these codelets are considered for recomposition, exhibiting good enough performance. At last, only 7 different compositions have been tried to find the best version of the function (which is the third composition we tested).

For the Hopping_Matrix loops, over four hundred codelets have been generated. A hundred of codelets have been discarded, leaving more than three hundred for evaluation. Most of these codelets exhibit very poor performance, and only forty-eight codelets are considered for recomposition. These codelets are those containing a direction computation, or one data flow of a direction computation. These codelets offer two possibilities for each direction computation, and with eight directions on each loop, $2^1 6$ combinations are possible. The best version for this function is the combination of all best stand-alone codelets, being the codelets with only one data flow of a direction computation. A dozen of combinations have been tried to verify that the recomposition of all the best stand-alone codelet is the best possibility.

After a peformance prediction far from encouraging, the code is executed with the new function. Surprisingly, the measured performance is better, with a 13% speed-up, as shown by the cluster of bars labelled "LQCD" in Fig.2.24. After further study, it appears that the optimization applied on the loop improving the data locality allows other function in the simulation to access more freely some of the data produced by the optimized function. Facilitating data reuse between the different functions of the iterative simulation brings more speed-up than the optimized hot-spot function itself. This behaviour cannot be predicted with the CPC method, since it focuses only on the hot-spot function (or loop). Interactions between functions are not considered, and the modification of their relations due to the optimization of the hot-spot function is unknown.

2.6 Conclusion

The approach presented in this chapter proposes to address long execution times of adaptive compilation. The goal of the CPC framework is to decompose the code into high performance codelets tested without considering the application context, and to recompose an optimized version of the code, according to a performance prediction model. We have shown on three real applications, cumulating four benchmarks, the validity of the approach. From the performance evaluation of these extracted and transformed codelets, a simple performance model is able to predict with accuracy (within a 10% range) the performance of the optimized code, if the data are in the correct memory hierarchy. A summary of the obtained predictions and measured speedups is displayed in Fig.2.24. The main advantage of the CPC framework is to allow the optimization of simulation programs without having to run them to test every performed transformations. However, the optimization space can be huge, and we resort on the use of pragmas to drive more easily the transformations application.

The hot-spot selection for applying optimizations is automatically done with code profiling programs. The extraction and the transformations are performed with X-language, requiring for the user to define the search space with directives inserted in the code. The selection of the codelets to evaluate, and the selection of efficient codelets to recompose the optimized function, are automatically done, the first on by analyzing the code with the MAQAO tool, the second through performance analysis. The actual recomposition is performed by the user, the prediction performance being computed automatically once the user specifies which recomposition is tested.

While many codelets are executed for the evaluation process, there is no need for multiple long execution of the whole application. With this method, the whole program is run only three times : two times before applying the framework to profile and benchmark the program (probes implanted in the code may interfere with the measure of the other study if done simultaneously), and only once with the new optimized function. The remaining evaluations are done only on extracted parts of the program, whether it is on the hot-spot function or directly on codelets. Executing all the required codelets still takes a negligible amount of time compared to applying a straightforward adaptive compilation.

Some gaps between predicted and measured performance can be seen in 2.24. The first gap, occurring for the LQCD benchmark, is due to unexpected inter-procedural optimizations. As the codelets are extracted and evaluated with no knowledge of the original program context, the transformations applied on the hot-spot function may

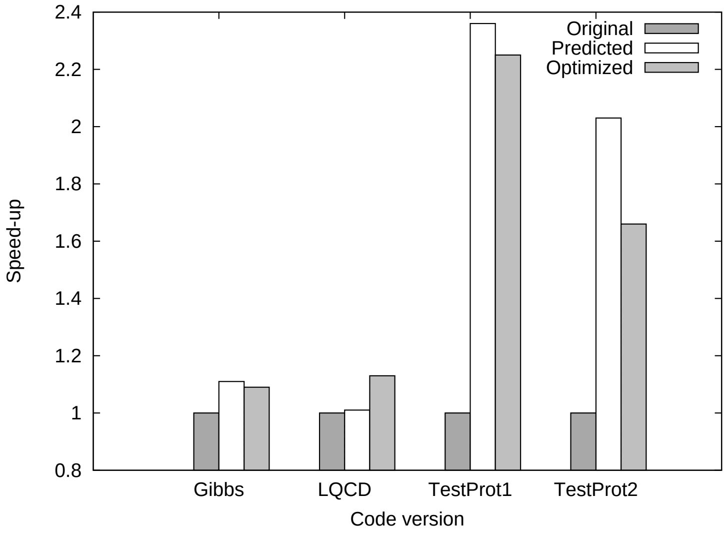


FIGURE 2.24 – Obtained speed-up on all presented benchmarks.

interfere with other parts of the code consuming data produced by the function. For this function, a tiling is applied, modifying the data locality of the produced results. The consumer of these results directly follows the optimized function, and benefits from the new data locality. The speedup due to this behaviour can not be evaluated in the framework, since all in the original program but the hot-spot function is ignored. The second gap visible on the "TestProt2" benchmark, is the result of our simple memory model. The data are assumed to be in a memory hierarchy, hence having enough element in the memory hierarchy to exhibit its peak performance, without exceeding it. With this benchmark, the amount of data is just slightly greater than the last level of cache. There is still a good data reuse, even if some data are in memory. With our simple model, we can consider only the two following cases : either all the data are in memory, or all the data fit in the last level of cache. If we consider that all data are in memory, the speedup due to cache reuse will not be measured, and the performance prediction will be greatly underestimated. On the other hand, considering that all data fit in the cache level, the latency of the data requests in memory is ignored, and the prediction is overestimated. Since the overestimated prediction is closer to the reality of data placement, the last solution was chosen, inducing the gap between the overestimated prediction and the actual performance measure.

These gaps offer opportunities for improvement on the prediction model. The results on the LQCD benchmark suggest to extract not only the hot-spot function, but also the following function in the running program. This "support" function will not be optimized, but will be benchmarked as a meta-codelet, also multiple data placement in the memory hierarchy. With this evaluation, the interference of the transformations with the whole program can be measured, and the prediction will be better. The second gap calls for a more precise memory model. Either several other memory placements are tested (for example, data filling a quarter, a half or three quarter of a cache level), or a codelet could be evaluated with a memory locality consistent to the one it will encounter in the whole program. Then a memory trace should be performed, which is time and memory consuming.

Another improvement opportunity lies with the optimization search. Still guided by the user, it could be fully automated with the integration of a search model base of the characteristics of the codelets. For example, while the optimization space is swapped, MAQAO can check the quality of codelets in a regular basis. If the quality of the produced code is too low, then the optimization space is pruned of further investigation in this direction. The absence of this automatic behaviour is the main interference toward its inclusion into an optimizing compiler. On the other end, the optimization space can be guided by the middle end of a compiler. This "codeleting" could be the reverse of the inline optimization.

Chapitre 3

Auto-tuning on stencil computation codes

In this chapter is presented a new approach to generate automaticaly an efficient data layout for multithreaded stencil codes on CPUs and GPUs. The concept of usual padding is extended to allow a better flexibility to address stream alignment conflicts. Alignment issues for efficient vectorization and concurrent simultaneous memory accesses are discussed. The work presented in this chapter is currently submitted for publication in the ICS 2012 conference.

3.1 Introduction

Stencil based computations represent a large class of applications, ranging from image processing, computational electromagnetics, hydrodynamics, lattice QCD or other physics simulations requiring the resolution of PDEs using finite difference or volume discretization. However, the variety of stencil kernels used in practice make this computation pattern difficult to assemble into a high performance computing library. Besides, the low flop/byte ratio that most common stencil exhibit requires to precisely tune the data layout and optimize memory accesses according to the architecture features.

The petaflop era has given rise to architectures of increasing complexity. Modern architectures combine many different levels of parallelism and a large memory hierarchies. SIMD instructions, such as those proposed in Intel SSE and AVX ISA for instance, and multi-thread programming offer opportunities to use this parallelism to reach high level performance. This comes however at the cost of a careful data layout organization in order to match memory alignment constraints. Combining vectorization and memory bank conflicts limitation with a proper data layout is a key to performance in current multicores and GPUs.

Automatic transformations for every stencil pattern is important, as the variety of stencil kernels is very large. Generating efficient code for CPUs and GPUs, taking into account alignment requirements, is paramount. Stream alignment conflicts are a fundamental algorithmic issues, as shown by Henretty *et al.* [52].

In this chapter, we develop a novel strategy to automatically generate stencil code for CPUs and GPUs, searching for the best data layout to answer alignment issues. We introduce a new data layout transformation, called *multi-padding*, extending the usual padding so as to maximize the number of aligned loads for vectorization. We present several methods to find the best paddings, with different levels of complexity. We show on stencil codes, in particular Jacobi and Laplacian computations, that generated codes compare well with hand-tuned codes, and that multi-padding can bring significant performance gains compared to the usual padding.

This chapter is organized as follows. The problematic of alignment conflicts is explained in section 3.2. In section 3.3, the vectorization on stencil codes is presented on a Jacobi 2D stencil, first with an example with a perfect behaviour, then with misalignment occurring. A method to find the best padding is shown, with the idea of having multiple padding values. This method is expanded in section 3.4 on a Jacobi stencil with more than two dimensions, introducing the concept of *Multi-dimensional padding.* Then, based on this case study, a formulation to align efficiently any stencil pattern (called *Multi-dimensional Multi-padding*, or **MDMP**) is developed in section 3.5. Using this formulation, a common strategy to automatically generate stencil code (with any patterns) for CPUs and GPUs, searching for the best padding possible to answer alignment problems, and taking into account the memory hierarchy of the target architecture is shown in section 3.6. The novelty of the approach is discussed in section 3.7, before presenting the performance mesures in section 3.8.

3.1.1 Stencil computation

A computation is called a "stencil computation" if it involves the update of elements in an array, with the update requiring several elements in the same array. Stencil computations are mainly used to simulate a propagation in a discretized finite space. As an example, consider a stencil computation using two n-dimensional arrays $(A[V_1, \ldots, V_n]$, $B[V_1, \ldots, V_n]$ with the form :

$$\ldots = f(A[i_1 + d_{11}, \ldots, i_n + d_{1n}], \ldots, A[i_1 + d_{m1}, \ldots, i_n + d_{mn}], \ldots$$

$$\ldots, B[i_1 + \delta 11, \ldots, i_n + \delta 1n], \ldots, B[i_1 + \delta p1, \ldots, i_n + \delta pn], \ldots)$$

To update an element, this computation uses m elements in the matrix A, and p elements in the matrix B. All elements of A (resp. B) belongs to an access pattern, defined by $(i_1, \ldots, i_n)$ and the matrix $(d_{hk})_{hk}$ (resp. $(\delta_{hk})_{hk}$).

As another example, Fig.3.1 presents a stencil pattern in a 3D space, used in earth sciences for seismic wave propagation.

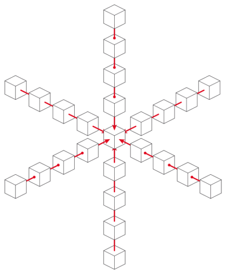


Figure 3.1 – An example of stencil pattern used in earth sciences

3.1.2 Compact Stencil

The strict definition of a compact stencil is a type of stencil using all nodes being direct neighbors, i.e. inside the convex envelop surrounding the updated cell in a distance of one element (9-point stencil in 2D, 27-point stencil in 3D). Usually, stencil pattern using only some nodes in this convex envelop are also called compact stencil (as the Jacobi 2D stencil pattern used in our case study).

3.1.3 Selfish Stencil (No weight on central node)

Usually, to update a cell in a stencil, one requires the values of the neighbour-cells, with the value of the current cell being updated.

$$B[i][j] = c.B[i][j] + c_{-i}.B[i-1][j] + c_{+i}.B[i+1][j] + c_{-j}.B[i][j-1] + c_{+j}.B[i][j+1]$$

FIGURE 3.2 – pseudo-code of a 2D Laplacian five-point stencil

In Fig.3.2 is an example of a classic 2D five-point stencil. We define a "Selfish Stencil" to be a stencil which, for each cells being updated in the computation, only requires the values of the neighbour-cells, and not its own value.

$$a_t[i][j] = c_{-i}.a_{t-1}[i-1][j] + c_{+i}.a_{t-1}[i+1][j] + c_{-j}.a_{t-1}[i][j-1] + c_{-j}.a_{t-1}[i][j+1]$$

FIGURE 3.3 – pseudo-code of a 2D Jacobi four-point stencil

Therefore, the "selfish" version of the 2D five-point stencil, presented in Fig.3.2, will be 2D four-point stencil. This four-point stencil will be the same as the five-point without the first member of the computation, as shown in Fig.3.3. It is the representation of Jacobi iteration method

3.1.4 Hollow shell

In order to avoid special code for cells on the boundaries of the stencil, thus creating conditions and non uniform control flow, we resort to the use of a "hollow shell", wrapping the computed stencil in a one-cell wide shell of null cells. Hence, we can apply the exact same computation for the cells on the borders of the stencil than for those inside the stencil. If a cell which does not exist on the original stencil is accessed, its null value will prevent it from impacting the result of the computation, producing the same result as if we had made a special case to only access the originally existing values. If the stencil code is vectorized, the hollow shell is extended to a vector wide shell of null cells.

3.2 Efficient Data Layout

In the last decade we observed a paradigm change in High Performance Computing, as higher computing capabilities moved away from exponential scaling of clock frequency toward chip multiprocessors. This change caused the need for more memory management, with the apparition of shared caches between multiple processors, and the possibility for several threads to read and write the same memory data at the same time. Moreover, multicore systems are not limited to multicore CPUs, but also include a collection of hardware accelerators, some of which are widely used, like GPUs. Bank conflicts are an issue in all multicore systems, as they decrease the amount of parallelism for memory accesses and add latency to memory transactions. For multicore architectures supporting hardware vectorization, one has to deal with a set of entangled hardware constraints, both coming from vector memory accesses and from multi-threaded memory accesses.

3.2.1 Set of simultaneous threads

In modern CPUs and GPUs architectures, a set of threads can run simultaneously on the same chip. To simplify, we will use the term *warp* to refer to such set of simultaneous threads. A warp is a basic notion in GPU programming. We extend it to multicore CPUs, considering a warp as the collection of threads running on a single chip, with a maximum of one thread per core (no hyperthreading is considered). On GPU, due to the SIMD nature of the architecture, threads of a same warp are synchronous. On CPU, there is no such garantee.

Vertical parallelization consists in slicing an iteration space in several parts, which will be swapped by a thread, and is used very often to parallelize linear algebra or stencil codes. It implies that simultaneous threads perform the same instruction at nearly same time, issuing multiple memory requests simultaneously. Whenever these requests may target the same memory bank, this is called a bank conflict. Memory accesses to the same bank are then serialized instead of being executed in parallel (for different banks). Requests on a busy memory bank will be delayed, and memory accesses, which are often already the limiting factor in stencil codes, will take even more time than usual to fetch the required data.

To avoid this problem, the data accessed by threads must have a different memory alignment modulo the number of memory banks. However, warp size can be greater than the number of memory banks. In this case, it is better to divide memory requests equally between all memory banks, than to aggregate numerous requests on the same bank, thus greatly increasing the memory latency. Since each threads will follow the same pattern of memory access, once the firsts simultaneous requests are equally spread across all available banks, all simultaneous data transactions will be issued on separate memory banks.

3.2.2 SIMD Memory Accesses

In order to obtain high performance and tip the balance between memory accesses and computation, array accesses have to be vectorized. Vectorization is detailed in the introduction, with algorithmic vectorization being described in section , and details on architectural vectorizarion being displayed in section . So far, many architectures (including Intel AVX) exhibit different performance depending on whether the accesses are align or unaligned on a x byte boundary, with x the size of the vector. A vector is composed of several elements, depending on the element size and most vector operations are element-wise (this changes in recent vector ISA). For a stencil computation, all accesses to the cells of the stencil pattern must be vectorized. And for the update to be correct, each neighbour must be at the same index in their respective vector (i.e *aligned*).

Vectorizing any computation consists in loading the elements required for the computation at the same index in their respective vector registers. Two different cases can occur : the element is said to be *naturally aligned* to its index, or misaligned. If it is misaligned, then the mechanism used to load the value in the register will be more time consuming than if it is naturally aligned. The main methods to align misaligned elements are :

- **Misaligned Load** : Load data from an unnatural alignment in a vector register. On many architectures, this comes at the expense of a performance penalty.
- **Memory Duplication** : Duplicate as many times as there are different alignment between required neighbours the elements array. Each of these arrays will be naturally aligned on one of the different alignment. Not to be used on memory bound computations
- **Shuffle** : Create with two naturally aligned vectors a third vector with the required elements. This mechanism add instructions and increase the register pressure. Shuffling is necessary for all elements of the pattern with row in Equation 3.16 that are not 0. With enough registers available, good register reuse, and a memory latency hiding the time of shuffles, shuffle can have no overhead at all.

- **Padding** : increases the size of the strides so as to align in memory elements of the pattern.

While the three first are time consuming or consume significantly more memory, padding has the advantage of only requiring a minor change in the data layout. No additional instructions are inserted in the code and, although there is an extra memory consumption, it is negligible compared to the Memory Duplication method. We explore in the following such technique for N-dimensional data layout. If the padding values are not enough to align all required data, then shuffles or unaligned loads are used, depending on the cost model of the target machine.

Vectorizing on GPUs

At first sight, GPUs do not seem to have this problem, since an element is composed with multiple registers, whereas on CPUs, a vector register is composed with several elements, causing alignment issue. On GPUs, a vector register always includes only one element. In CUDA, built-in vector types are proposed, providing programmers with a set of vectors of different sizes, up to 128 bits or a vector of four elements. In the CUDA Programming Guide [31], these built-in types are said to enforce specific alignment of data, sometimes with vector requirements being different from its base type requirement. Among others, a char1 must be aligned on a 8 bits address, char2 on a 16 bits, char3 on a8 bits address, char 4 on a 32 bits address. Using these built-in types will cause the same alignment issues than for CPU.

Compulsory alignment conflicts

Padding transformation sets a dimension in an array to a new size. It adds a specific number of cells at the end of a line, to change in memory the beginning address of the next row. As it can only modified alignments between to lines, alignment conflicts occurring in the same line will not be affected. These conflicts are called *compulsory alignment conflicts*. On the example of the Jacobi 2D stencil with a vector of size 4, displayed in Fig.3.5, E and G cannot be aligned on the same value. A *compulsory alignment conflicts* exists between the two elements.

3.3 Jacobi 2D case study

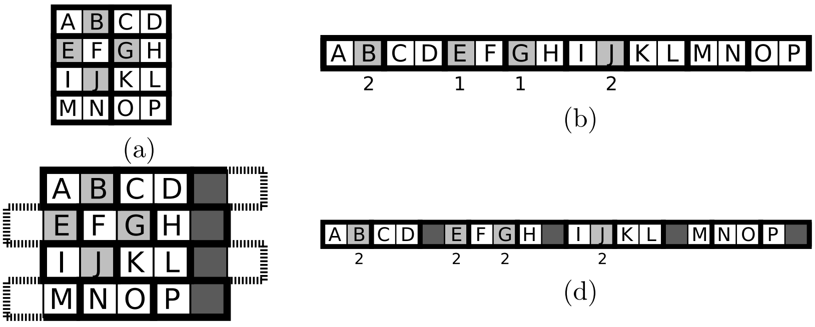
In this section, a Jacobi 2D stencil is considered. The form of this four-point 2D stencil is :

$$A[i][j] = c_{-i}.B[i-1][j] + c_{+i}.B[i+1][j] + c_{-j}.B[i][j-1] + c_{-j}.B[i][j+1]$$

Fig.3.4**(a)** represents the pattern of this stencil, with the grey cells being those required for the update of the central cell. Here, cells B, E, G and J are necessary to update cell F.

In order to vectorize efficiently this stencil computation, padding cells will be used to align the maximum of required elements to the same alignment. A first example, with vector composed of two elements, has a perfect behaviour. A simple adding is enough to align all required cells, as detailed in section 3.3.1. For the second example, in section 3.3.3, *Multi-padding* is introduced, which consists in having the possibility of have multiple padding values, circling in a periodic way.

3.3.1 Detection of necessary padding, with vector of two elements


A B C D
E F G H
I J K L
M N O P
A B C D E F G H I J K L M N O P
2 1 1 2
A B C D E F G H I J K L M N O P
2 2 2 2

(a) (b) (c) (d)

FIGURE 3.4 – Modification of element alignment in memory with a simple padding on a 2D Jacobi stencil computation with a vector size of two elements.**(a)** Vectorized Jacobi 2D with no padding. **(b)** Vectorized Jacobi 2D with no padding, memory view. **(c)** Vectorized Jacobi 2D with simple padding = 1.**(d)** Vectorized Jacobi 2D with simple padding = 1, memory view.

In our 4-point 2D-Stencil case, we need for all four elements $B[i-1][j]$, $B[i+1][j]$, $B[i][j+1]$ and $B[i][j-1]$ (elements B, E, G and J in Fig.3.4**(a)**. to be on the same alignment in order to vectorize efficiently. Having the four elements on the same alignment will allow to directly consider vectors, as long as the data are contiguous, whereas different alignments will force to use different mechanisms in order to build the vectors for the computation. These mechanisms will have an impact either on the memory print (additional arrays being created for the previously misaligned data to be correctly aligned), or on the instruction print (insertion of shuffle instructions to build the needed vector from previously worthless vectors). Therefore, these mechanisms will reduce the overall performance and should be avoided if possible.

In the double precision case, this condition will be rather easy to satisfy.

First, we can easily establish that in this particular stencil case, the two cells needed in each direction to compute one point of the Stencil have the same alignment. Assuming that the size of the outer dimension of our 2D array A is S_j, and that the address of $A[i, j]$ is defined by $A[i, j] = iS_j + j$, then the two elements needed in the same row have the same alignment if :

$$\forall i, \forall j, iS_j + j - 1 \equiv_2 iS_j + j + 1$$

Since the distance between the two cells is two elements, they will always have the same congruence modulo 2. For the two elements needed in a column, they have the same alignment if :

$$\forall i, \forall j, (i-1)S_j + j \equiv_2 (i+1)S_j + j + 1$$

Here again, the distance between the two cells is $2S_j$, hence a multiple of 2. The two elements will always have the same alignment.

According to this result, if one manages to have one cell in each direction with the same alignment, the whole array will be well aligned. The vectorization of the computation will then be a rather simple transformation to apply, No other code transformations will have to be used to ensure the correctness of the produced vectors, and of the new code.

To search for a possible common alignment, elements $a[i+1][j]$ and $a[i][j+1]$ were selected. Their respective addresses compose the system of equations :

$$\forall i, \forall j, \left.\begin{array}{r} i * S_j + (j+1) \equiv_2 c \\ (i+1) * S_j + j \equiv_2 c \end{array}\right\} \tag{3.1}$$

The resolution of equation 3.1 shows that the S_j must be odd for the vectorization to be applied efficiently.

Furthermore, it implied that a padding must be done if the line size is even. When the number of columns of our 2D-Stencil is originally even, the equation 3.1 forces a padding by an odd number of element for the whole array to be well aligned.

Considering the example in Fig.3.4, the original size of a line was even (4 elements in Fig.3.4(**a**)), and the four required elements for the update of F were not on the same alignment (Fig.3.4(**b**)). Then a padding of one cell has been applied, going from a size of 4 elements to a line size of five elements (Fig.3.4(**c**)). With this new line size, all required data are aligned on the same value (as shown in Fig.3.4(**d**)).

Unfortunately, when applying some transformations on a code, problems can appear when using the transformed elements. Vectorization can have the same limitation. To ensure that we can consider the transformed array as a whole, one have to check if all elements will behave correctly during the computation. If some elements are not, we can separate them from the suitable ones, and isolate them so that they will not interfere. We will then consider on one hand good vectorized elements, and then deal with the others (applying the original sequential computation on them, or deleting values written in the hollow shell).

3.3.2 Vectorization boundaries, using vector of two elements

If the line size is not a multiple of the vector size, or when applying some padding, a line boundaries may not correspond to vectors boundaries. It means that a vector being updated can be composed with elements of the hollow shell. These null elements will be written with some values, useless for the stencil computation, compromising the integrity of the hollow shell. There are two possible ways for these elements to remain with a null value at the end of the update. On one hand, the elements are not computed in a vectorized way, and the original sequential code is used for the update on these elements. On the other hand, the update of the elements is vectorized, then a procedure will delete the wrong values. One has to know which elements are not naturally eligible for vectorization, either to update these elements sequentially, or to delete the values written in the hollow shell. In this section, we describe how to find elements to compute sequentially. Note that it is easy to establish, if the other solution is considered, which cells have to be cleared, since the elements to delete are the complementary elements of the ones computed sequentially in a vector. For instance, with vector composed of 8 elements, if 5 cells should be updated sequentially, then 3 cells (8-5) must be cleared if the update is performed in a vectorized way.

In the previous section, we have ensured that all elements were correctly aligned in the 2D-array. Since a vector is composed of two values, here two cells of the array, all even (or odd) elements of the array have the same alignment. To simplify, we will consider that the first element of the computed stencil is the first part of the first vector. Hence, every even element of the stencil will be the first part of a vector, and every odd element the second part of a vector. Knowing this, we can identify which cells in the array are not eligible for vectorized computation.

A vector is supposed to be composed with two contiguous double values. However, it can happen that the two values, contiguous in memory, are not contiguous in the array. As an example, the last element of a row and the first element of the next row in an array are not strictly contiguous. However, it is possible that these elements independent in the array are contiguous in memory. If one want to vectorize this array, it may be possible that the last element of a row and the first element of the next row will compose a unique vector. This vector is not a correct vector, and to use it in some computation will lead to mistakes in results.

As to rid the vectorized space of these cases, one as to determine if a first cell of a row is not the first part of a vector, and also if a last cell of a row is not the last part of a vector. Even if this kind of behaviour disappear with the application of a hollow shell wrapping the array (as described in section 3.1.4), the problem it raised remains : we have to check if there is first element of rows which are not "even" cells, and lasts elements of rows which are not "odd" cells.

The modifications brought in the previous section to align all required data ensure that these special cases happen. Dark grey cells and dotted lines in Fig.3.4**(c)** show these cases.

Let us consider that S_j is the original line size, and S_p is the new padded line size (S_p is an odd value).

The equation verifying that the first computed element in a row is at the second place in a vector is :

$$\forall i, i * S_p + 0 \equiv_2 1$$

Since the new line size S_p is odd, this equation demonstrates that for any original sizes of rows, the first cell of even rows in the original stencil will never be eligible to form a usable vector.

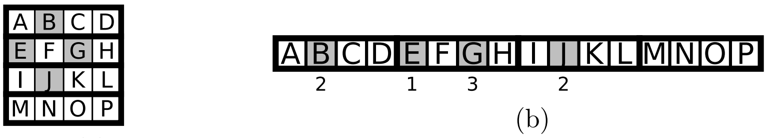
However, the original size of rows has an impact on the location of the non-vectorizable last cell on the stencil rows. The equation verifying that the last computed element in a row is at the first position in a vector is :

$$i * S_p + S_j \equiv_2 0$$

According to its resolution, if the original row size is odd, last cell of each odd rows in the stencil will be the special ones, whereas if the original row size is even, the last cell of even rows will be the ones to take care of. The example in Fig.3.4**(c)** illustrates the last case. The original line size is even (four elements), and the cells not eligible for vectorized computation are at the end of lines 0 and 2 (even rows), and at the beginning of lines 1 and 3 (odd rows).

3.3.3 Detection of necessary padding, with vector of any size

Now, we will apply the same analysis as the two last sections on the same stencil pattern, but this time vectors are composed with more elements than two.



(a)

(b)

FIGURE 3.5 – Representation of a Jacobi 2D pattern, using vector of four elements (simple data in SSE). **(a)** Logical view, update of F requires elements B, E, G, J. **(b)** Memory view, the four elements are not in the same place in vectors. E is in the first position, B and J are both in the second position, and G is in the third. Choosing alignment of B and J as the valid one, two shuffles or unaligned loads are required.

Fig.3.5**(a)** presents the same pattern of access as Fig.3.4**(a)**. The stencil uses the elements B, E, G, J, but with a larger vector size (4 in this example). Fig. 3.5**(b)** shows the offsets of these elements in the vectors of 4 elements (A, B, C, D), (E, F, G, H) and (I, J, K, L). Here again, the elements are not in the same positions in the vectors. One of the main changes between having vector of any size, compare to vector composed of two element, is that in our 2D Jacobi based case study, the two elements needed for the computation on the same line will not be well aligned at the same time. Considering that S_j is still the line size, and v the vector size, the system of equations 3.2, composed with the addresses of the two elements in the same line, is unsolvable.

$$\forall i, \forall j, \left.\begin{array}{r} i * S_j + j + 1 \equiv_v c \\ i * S_j + j - 1 \equiv_v c \end{array}\right\} \tag{3.2}$$

The first part of the system will quickly return $i * S_j + j \equiv_v v - 1$, whereas the second part will give $i * S_j + j \equiv_v 1$, the two solutions being incompatible if $v \neq 2$. It is an example of *compulsory conflict* described in section 3.2.2.

As a result, all the elements can not be well aligned for the vectorization. However, we will try to align a maximum of elements to improve the efficiency of vectorization the best we can. We then have to choose which one of the two needed elements in a row we will try to align to the elements needed in a column.

Until further notice, we choose to align the first needed element of a line to the other elements needed in the computation. When trying to align the required element on top of the updated cell ($a[i-1][j]$), with the required element on its left ($a[i][j+1]$), we end up with the system :

$$\forall i, \forall j, \left.\begin{array}{r} (i-1) * S_j + j \equiv_v c \\ i * S_j + (j-1) \equiv_v c \end{array}\right\} \tag{3.3}$$

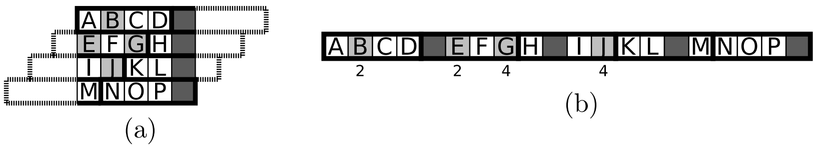
The resolution of this equation returns that $S_j \equiv_v 1$.

Now, let us suppose that this system is not the one considered. We build instead the system for the required elements on its left ($a[i][j-1]$), and the one under the updated cell ($a[i+1][j]$) :

$$\forall i, \forall j, \left.\begin{array}{r} i * S_j + (j-1) \equiv_v c \\ (i+1) * S_j + j \equiv_v c \end{array}\right\} \tag{3.4}$$

The resolution of this equation returns that $S_j \equiv_v v - 1$.

Unfortunately, the resolution of these two systems returns two different values.



(a)

(b)

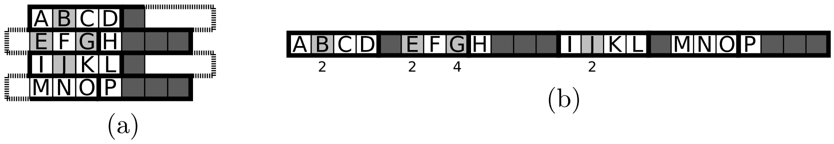
FIGURE 3.6 – Simple padding applied on the Jacobi 2D pattern with vector of four elements. **(a)** Logical view, alignment of elements have changed. Dotted lines represent unfinished vectors on a line, being completed with elements of the next line.**(b)** Memory view, two distinct alignments. Do not improve compared to no padding, since here again two shuffles or unaligned loads are required.

We ended up with different values for the same line size, proving that a simple padding cannot resolve the alignment problem with any vector size, unlike the simpler case studied in section 3.3.1. Since we ended up with two different values, we suggest to consider the same systems, but with two distinct line sizes, one for the even rows (S_e), and one for the odd rows (S_o) in the stencil array. System 3.4 is modified in system 3.5, while system 3.3 becomes system 3.6.

$$\forall i, \forall j, \left.\begin{aligned} 2i * (S_e + S_o) + (j+1) \equiv_v c \\ (2i+1) * S_e + 2i * S_o + j \equiv_v c \end{aligned}\right\} \tag{3.5}$$

$$\forall i, \forall j, \left.\begin{aligned} (2i) * S_e + (2i-1) * S_o + j \equiv_v c \\ (2i) * (S_e + S_o) + (j+1) \equiv_v c \end{aligned}\right\} \tag{3.6}$$

The new systems of equations have one common solution. Each line size gets only one possibility. Hence, in our 2D Jacobi stencil with a vector size of v elements, for a maximum of elements required in an update, the even rows should be padded for their size to be congruent to 1 modulo v, and the odd rows should be padded for their size to be congruent to $v-1$ modulo v. This observation leads to the concept of *Multi-padding*, which is originally the possibility to have a different padding value for each row of the computed array.



(a)

(b)

FIGURE 3.7 – Multipadding 1-3 applied on the Jacobi 2D pattern with vector of four elements. **(a)** Logical view, alignment of elements have changed. **(b)** Memory view, still two distinct alignment.B, E and J are in the same position in vectors. Now only one shuffle or unaligned load required.

In this kind of modular arithmetic, numerous values could verify these conditions. However, since padding values are cells added in the array, one will try for the sum of added cells to be as small as possible, limiting the impact on the memory consumption. Padding values must remain positive, otherwise the padding will be deleting useful cell from the computed array. With these new constraints, the values for S_o and S_e are easily found :

$S_e = 1$ and $S_o = v - 1$. Figures 3.7**(a)** and 3.7**(b)** are a perfect example. The vector size of four elements leads to a padding value for even rows of 1, and a padding value for odd rows of 3. With these sizes, a maximum of three required elements(B, E and J) are on the same alignment for the update of J. Only G is not correctly aligned, and will never be, due to equation 3.2.

Offset Value

To relax index computations, we wish for sizes for odd and even rows to be the same. Unfortunately, the conditions for S_o and S_e sizes do not allow them to have the same value if the vector is larger than two elements.

Since elements are contiguous in memory, it is possible to add an *offset* in the computation to pretend to move the border between two lines. This *offset* value has no impact on memory. It is a software trick for the programmer to simulate the use of only on line size, instead of the two distinct sizes found previously.

To check the impact of the simulation, a previous equation verifying the sizes for a common alignment was considered. The system 3.6 is modified to take into account the moving of cells from even rows to odd rows, by adding the value *offset* in the right spot. Equation 3.7 (resp. 3.8) is the modification of the first (resp. last) equation in system 3.6

$$\forall i, \forall j, (2i) * (S_e + offset) + (2i-1) * (S_o - offset) - offset + j \equiv_v c \tag{3.7}$$

$$\forall i, \forall j, (2i) * (S_e + offset + S_o - offset) + (j + 1) \equiv_v c \tag{3.8}$$

In equation 3.8, the *offset* terms appear naturally, and cancel each other quickly. Since the border is supposed to change between a couple composed with an even line and an odd line, and that the array is considered to begin with an even line, the moving cells are considered to be remove from the beginning of odd lines to be added at the end of their corresponding even ones. Then, in equation 3.7, the *offset* value must be subtracted from j, since it is on an odd row and *offset* cells have been removed at its beginning. Again, all *offset* values in the equation cancelled each other, retrieving the exact same equations as in the system 3.6. Equations 3.7 and 3.8 prove that removing cells of the end of even rows to put them at the beginning of the following odd rows does not change the alignment of the useful cells, since adding *offset* in the original equation has no impact. However, an implicit constraint appears. Since we are "taking" cells from the beginning of odd lines to put them at the end of even lines, we implicitly suppose that there is an even number of lines (as many odd lines than even lines).

To find the *offset* value, the equation Fig.3.9 is considered.

$$S_e + offset = S_o - offset \tag{3.9}$$

The solution of this equation is $offset = \frac{v-2}{2}$, here again with an implicit additional constraint. If the vector is composed of an odd number of elements, then we are supposed to have a simulated line size with half of a cell. Since it is very difficult to consider, vectors are supposed to be composed of an even number of elements.

To correctly use an *offset* value, to simulate only one line size, the 2D array must have an even number of lines, and the vectorization must take place with vectors composed of an even number of element.

3.3.4 Vectorization boundaries using vector of any size

As seen in Section 3.3.2, some cells in the stencil may not be eligible for vectorized computation. The larger size of vector widened the possibility of errors. In Section 3.3.2, vectors were build with two elements, hence having one possibility over two for the element to be misplaced in the vector, and the same probability to have a good placement in the vector). With a larger vector size, it is the same probability for a good placement (one over the size of the vector), leading to a larger probability of having a misplaced element.

$$\forall i, 2i * (S_e + S_o) \equiv_v 0 \tag{3.10}$$

To find and isolate non-vectorizable cells, one has to search for first element of lines which are not the beginning of a vector, or the last cell of lines which are not the last element of a vector. Equations 3.10 and 3.11 evaluates which are the vectorizable elements in lines beginning, deducing the not vectorizable ones, for even and odd lines. With $S_e = 1$ and $S_o = v - 1$, equation 3.10 will always be true. First cells of even rows will always be considered as vectors during the computation.

$$\forall i, (2i + 1) * S_e + 2i * S_o \equiv_v 0 \tag{3.11}$$

However, trying to solve equation 3.11 ends with $1 \equiv_v 0$, which is not possible. First cell element of odd lines will be always misaligned for a valid vectorized computation. Worse, due to its position (second element in the vector), the $v - 1$ firsts elements of every odd lines will not be eligible for vectorization.

$$\forall i, 2i * (S_e + S_o) + S_j - 1 \equiv_v v - 1 \tag{3.12}$$

$$\forall i, (2i+1) * S_e + 2i * S_o + S_j - 1 \equiv_v v - 1 \tag{3.13}$$

Equation 3.12 is true only if $S_j \equiv_v 0$, and equation 3.13 if $S_j \equiv_v v-1$. So, if the original line size S_j is congruent to 0 modulo the vector size, sequential cells will only be at the end of even rows. If the size is congruent to $v-1$, these cells will only be at the end of odd rows. For all other sizes, each row will found problematic vector at their end.

The position of the sequential elements depends on the cells considered when aligning data. When a compulsory conflict occurs, it means that there is at least two possible alignments for the elements on the line. One of these alignments is chosen, leading to a specific repartition of non-vectorizable cells. If another alignment is selected, the disposition of sequential computations is different.

3.3.5 Choosing between to equivalent alignment

In Section 3.3.3, a specific alignment was fixed for the stencil pattern : the first needed element of a row was chosen to be aligned to the other needed elements. This lead to have, in the beginning of each odd lines, seven cells requiring special treatment (see equation 3.11).

Between the two possibilities, maybe picking the first required element of the row was not the better choice.

In this section, the second needed element in a line will be aligned to the other needed elements (Fig.3.8).

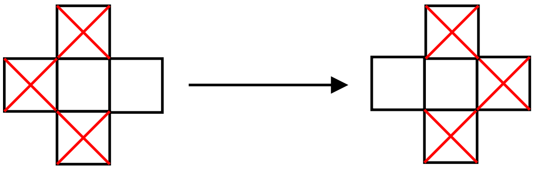


FIGURE 3.8 – Aligning the second needed element of a row

Doing the same kind of computation as in equations 3.4 and 3.3 but with the second needed element in a row, the new two following sizes were obtained for even and odd rows : $S_e \equiv_v v-1$ and $S_o \equiv_v 1$. This interchange between the two values of S_e and S_o will not impact the value of offset, computed in equation 3.9, nor the absence of problematic cells at the beginning of even lines, since, with these new values, $S_e + S_o \equiv_v 0$ as before in equation 3.10. Furthermore, the interchanged value will have only one impact on the conditions found with equation 3.12 and 3.13 : they also will be interchanged.

However, the new value of S_e will have a more important impact on the result of equation 3.11. To be a valid vector for the computation, the first element must be congruent to 0 modulo the size of the vector, and also to be in the original stencil. Hence, due to the choice of the cell in a row to be align, we ended up with $v-1$ sequential computations at the beginning of each odd rows (see equation 3.11). However, the alignment change, and therefore the interchange of S_e and S_o, will brought a better number of sequential cells.

With the new sizes, equation 3.11 will return a size of $v-1$ for S_j. The first element of odd rows are still not well aligned. However, its new alignment still have a great impact. Since it is misaligned, the first vector eligible for computation is the one beginning with the following cell having a good alignment. Unlike with the previous alignment, the next valid cell for being the first element of a vector is not $v-1$ cells further, but only one. Therefore, the second cell of an odd row will always be eligible for vectorized computation, and only one sequential computation at the beginning of an odd line will take place, instead of $v-1$ with our previous choice of aligned cells.

The choice of which cell of the line will be aligned depends on how the special cases will be treated. If one wants to do the whole computation with vectors, then delete data written in the hollow shell, the first alignment must be chosen, since it is the one with the less written cells in the hollow shell. Less time will be lost while erasing wrong values. On the other hand, if the other solution is selected, which is computing problematic cells sequentially, the new alignment tested in this section is the best. There will be less sequential computation, and the overall performance of the stencil computation will be improved.

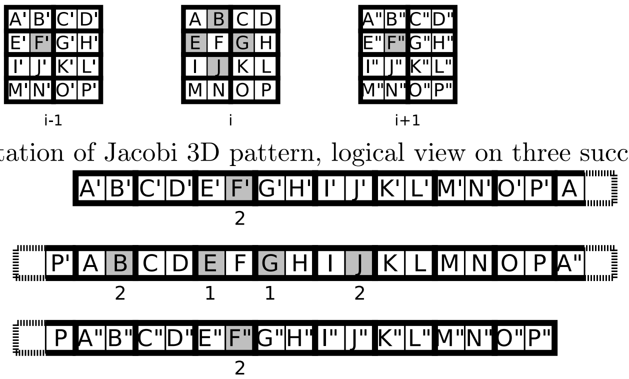
3.4 Jacobi N-D case study

In this section, we expand the work done in section 3.3 on a Jacobi stencil for any number of dimensions. Considering that n is the number of dimension,the formulation of the stencil is :

$$A[i_0,\ldots,i_{n-1}] = \sum_{k=0}^{n}(c_{-i_k}.B[i_0,\ldots,i_k-1,\ldots,i_{n-1}]+c_{+i_k}.B[i_0,\ldots,i_k+1,\ldots,i_{n-1}])$$

Fig.3.9 offers a visual example of the Jacobi stencil pattern in 3D. The update of cell F requires elements F', B, E, G, J and F".

This section introduces two important notions : *Multi-dimensional padding* with the first example, where, with vector composed of two elements, a padding can be needed in each dimension to align all required elements. Then, *Multi-dimensional Multi-padding* (or **MDMP**), allowing multiple padding values on each dimension, is quickly presented.


A' B' C' D'
E' F' G' H'
I' J' K' L'
M' N' O' P'
i-1
A B C D
E F G H
I J K L
M N O P
i
A" B" C" D"
E" F" G" H"
I" J" K" L"
M" N" O" P"
i+1
A' B' C' D' E' F' G' H' I' J' K' L' M' N' O' P' A
2
P' A B C D E F G H I J K L M N O P A"
2 1 1 2
P A" B" C" D" E" F" G" H" I" J" K" L" M" N" O" P"
2

(a) Representation of Jacobi 3D pattern, logical view on three successive planes.

(b) Representation of Jacobi 3D pattern, logical view.

FIGURE 3.9 – Representation of a Jacobi 3D pattern, using vector of two elements (double data in SSE). **(a)** Logical view, update of F requires elements B, E, G, J, F' and F". **(b)** Memory view, the three lines are contiguous in memory (P' in the first and second lines are the same one). The six elements are spread through the two possible alignment. Choosing alignment of B, J, F' and F" as the valid one, two shuffles or unaligned loads are required.

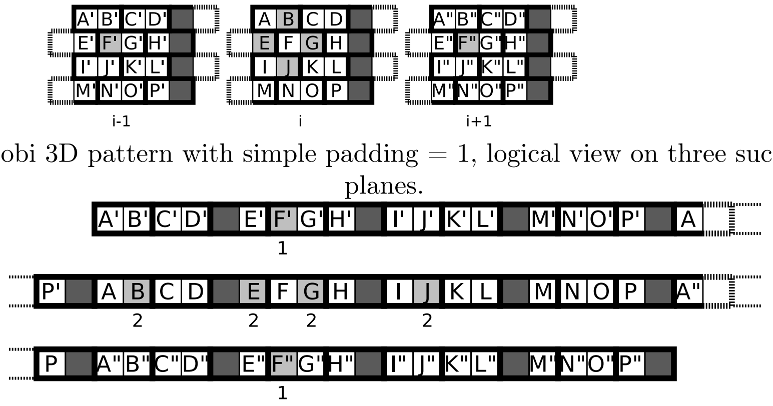
3.4.1 Detection of necessary padding, with vector of two elements

Based on the study for the Jacobi 2D in section 3.3.1, the possibility of a common alignment with all necessary elements for an update in a n-dimensional Jacobi stencil will be proved by induction.

First, let us show that the two needed elements in one direction will always have the same alignment, when the vector is composed with two elements. Considering that S_b is the size of the b^{th} dimension, and i_b the position of the element in the b^{th} dimension, equation 3.14 represents our hypothesis that $i_{k+1}+1$ and $i_{k+1}-1$, the two elements required in the $(k+1)^{th}$ dimension, have the same alignment, $\forall i_0, \ldots, i_{n-1}$.

$$\left.\begin{aligned} i_0 + \sum_{a=1}^{k}(i_a * \prod_{b=0}^{a-1} S_b) + (i_{k+1}+1) * \prod_{b=0}^{k} S_b + \sum_{a=k+2}^{n-1}(i_a * \prod_{b=0}^{a-1} S_b) \equiv_2 c \\ i_0 + \sum_{a=1}^{k}(i_a * \prod_{b=0}^{a-1} S_b) + (i_{k+1}-1) * \prod_{b=0}^{k} S_b + \sum_{a=k+2}^{n-1}(i_a * \prod_{b=0}^{a-1} S_b) \equiv_2 c \end{aligned}\right\} \tag{3.14}$$

Solving equation 3.14 ends up with an always true statement. It means that whatever the sizes of the dimensions are, the two elements required in a specific dimension to update a cell of a Jacobi stencil with n dimensions have always the same alignment. Fig.3.9**(b)** confirms that the couple of elements (E, G) in the inner-most dimension have the same alignment, as the couple (B, J) in the middle dimension, and the couple $(F', F")$ in the outer-most dimension.



(a) Jacobi 3D pattern with simple padding = 1, logical view on three successive planes.

(b) Jacobi 3D pattern with simple padding = 1, memory view.

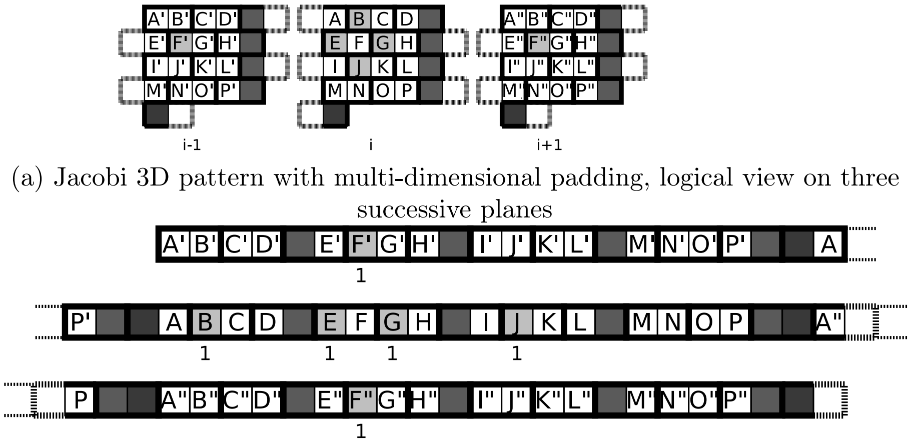
FIGURE 3.10 – Simple padding applied on the Jacobi 3D pattern with vector of two elements. **(a)** Logical view, alignment of elements have changed. Dotted lines represent unfinished vectors on a line, being completed with elements of the next line.**(b)** Memory view, two distinct alignments. Do not improve compared to no padding. There is again four elements on one alignment (second position), and two elements on the other possible alignment. Again, two shuffles or unaligned loads are required.

Now, like in the 2D case, if one manages to align one cell of each dimension, the complete n-dimensional array will be correctly align for a vectorized computation. A proof by induction will show that if the elements of the first k dimensions can have the same alignment, then the elements of the $(k+i)^{th}$ dimension is also on this alignment. Equation 3.15 represents this hypothesis, $\forall i_0, \ldots, i_{n-1}$.

$$\left.\begin{array}{l} i_0 + \sum_{a=1}^{k}(i_a * \prod_{b=0}^{a-1} S_b) + (i_{k+1} - 1) * \prod_{b=0}^{k} S_b + i_{k+2} * \prod_{b=0}^{k+1} S_b + \sum_{a=k+2}^{n-1} (i_a * \prod_{b=0}^{a-1} S_b) \equiv_2 c \\ i_0 + \sum_{a=1}^{k}(i_a * \prod_{b=0}^{a-1} S_b) + i_{k+1} * \prod_{b=0}^{k} S_b + (i_{k+2} - 1) * \prod_{b=0}^{k+1} S_b + \sum_{a=k+2}^{n-1} (i_a * \prod_{b=0}^{a-1} S_b) \equiv_2 c \end{array}\right\} \tag{3.15}$$

The resolution of the system 3.15 shows that if the $k+1^{th}$ dimension has an odd parity, the needed elements will have the same alignment.

With an odd size on the $k+1^{th}$ dimension, the elements of this dimension will be aligned as the ones of the previous dimension. Using the work in section 3.3.1 as the first step of our recurrence, it is induced that for all required elements for the computation of a cell in our stencil to be on the same alignment, for any number of dimensions, the sizes of all the dimensions (except for the outer-most dimension) must be odd. It introduces the concept of *Multi-dimensional padding*, as each dimension may have to be padded.



(a) Jacobi 3D pattern with multi-dimensional padding, logical view on three successive planes

(b) Jacobi 3D pattern with multi-dimensional padding, memory view

FIGURE 3.11 – Multi-dimensional padding applied on the Jacobi 3D pattern with vector of two elements. **(a)** Logical view, alignment of elements have changed. In addition to the simple padding, an extra element is added at the end of each planes.**(b)** Memory view, all six required elements have the same alignment (first position in vectors). Only aligned loads will be performed, no additional shuffles or unaligned loads are required.

Like in the example in fig.3.9**(a)**, if some dimensions have even sizes, then they must be padded to change their parity. In fig.3.9**(b)**, with an even size of 4 for all dimensions, the elements required for the update of F are not all aligned on the same value. A simple padding applied in fig.3.10does not solve the problem. Only the padding of each dimension to an odd size in fig.3.11allows all six elements F', B, E, G, J and F" to be at the same position in vector (here, the first place in their respective vector).

3.4.2 Vectorization boundaries with vector of two elements

This step happens to be the exact same procedure as in section 3.3.2. Due to the construction method of vectors, only the inner dimension (the rows of the stencil) has to be considered. Hence the constraints to find cells not naturally eligible for vectorized computation are exactly the same as in section 3.3.2.

3.4.3 Detection of necessary padding, with vector of any size

Coupling the results of section 3.3.3 with the results of section 3.4.1, we determine that, for a N-D Jacobi stencil using vector composed of more than two elements, a *Multi-dimensional Multi-padding* is necessary. Each dimension requires to have two different padding values, 1 and $v-1$, v being the size of a vector. The heavy equations demonstrating this result are displayed in Annex A.

This **MDMP** method will be formally described in section 3.5.

3.5 General Case of stencil computation

After focusing on the two studied cases, *Multi-dimensional padding* and ***MDMP*** methodologies are formally described in this section, for any stencil pattern. A new formulation using matrix representation is used to lightened equations. Compared to the section 3.4, the methodologies sustain a slight modification : rather than expand the dimension sizes with the required number of hyperplane, only several cells will be added at the end of a dimension. The first reason for this change is due to the me memory consumption. When an outer dimension is increased, it causes new hyperplanes to be stored in memory, taking a lot of place. Adding only few cells will have the same consequences on alignments, without greatly stressing the memory. Furthermore, since any stencil is considered, one must try to align all required elements, and not only a special selection, as it was done in sections 3.3 and 3.4 due to the regular nature of the pattern. It results in a complex system of equations, processed by a linear solver. If, instead of only adding a few cells at the end of a dimension, increasing the dimension sizes was kept, then the system to solve would not have been a linear one but a quadratic one, which is far more complex.

3.5.1 Formulation of alignment constraints

We study in the following sections a stencil computation on a n-dimensional array $A[V_1, \ldots, V_n]$ of the form :

$$\ldots = f(A[i_1 + d_{11}, \ldots, i_n + d_{1n}], \ldots, A[i_1 + d_{m1}, \ldots, i_n + d_{mn}])$$

This computation reads all elements of A belonging to an access pattern, defined by $(i_1, \ldots, i_n)$ and the matrix of integers $(d_{hk})_{hk}$. We assume in the rest of the paper that all elements of A are mapped contiguously in memory, and to simplify notations, we assume elements of A are 1 byte long.

Vectorizing such expression requires to load into SIMD registers, at the same position in the vectors, all the elements of the access pattern. Optimization of this code for GPU is discussed in Section 3.6.

We first discuss alignment issues for an SIMD implementation, illustrated on a Jacobi stencil, then introduce the multi-dimensional padding. Finally we extend it further into multi-dimensional, multi-padding.

In Section 3.3.3, Fig.3.5**(a)** presenting the original pattern access of a Jacobi 2D stencil computation show that B, E, G, J, the required elements for the update of

F are not naturally aligned. Shuffles or unaligned loads are needed to do a correct vector computations.

More generally, we propose to describe the conditions when the elements of a stencil pattern are all aligned : For SIMD vectors of size l, this implies that the addresses of these elements are the same modulo l. The memory address of an element $A[i_1, \dots, i_n]$ is defined by :

$$\&A[i_1, \dots, i_n] = A + \sum_{k=1}^{N} i_k S_k$$

where S_k define the strides separating consecutive elements of A in the k^{th} dimension. Thus, given a vector $(i_1, \dots, i_n)$, the elements of A required for the stencil computation have the same alignment if and only if :

$$\exists c, \forall h, A + \sum_{k=1}^{n} (i_k + d_{hk}) S_k \equiv_l c$$

where $\equiv_l$ is the identity modulo l and c is a constant.

Thus, all elements are aligned if and only if the vector of strides (S_k) check the following constraint :

$$\exists c, \begin{pmatrix} d_{11} & \dots & d_{1n} \\ \vdots & \ddots & \\ d_{m1} & & d_{mn} \end{pmatrix} \begin{pmatrix} S_1 \\ \vdots \\ S_n \end{pmatrix} \equiv_l \begin{pmatrix} c \\ \vdots \\ c \end{pmatrix} \tag{3.16}$$

When Equation (3.16) is not checked, some elements are misaligned in memory. Misaligned elements in memory would have to be aligned in registers for the computation to be correct. For instance, in the Jacobi 3D presented in Fig.3.9**(a)** (showing three planes of the volume) and **(b)** (linearized memory), the elements required by the computation (F', B, E, G, J, F'') have not the same alignment (resp. $(2, 2, 1, 1, 2, 2)$). E and G are unaligned compared to the other elements.

3.5.2 Formulation of Multi-dimensional Padding

Let us assume, with no loss of generality, that the array is row-major (as in C). We describe in this section a simple padding for a 2D array, and then generalize this to a padding for a multi-dimensional array.

A simple padding for a 2D array consists in adding elements at the end of each row, in order to align elements needed in a stencil pattern. The number of elements is the same for each row. For instance, Fig.3.6describe the access pattern of a Jacobi 2D when row are padded with one element. The elements B, E, G, J needed by the computation have two different alignments. While better than the first vectorization (with no padding, figure **(b)**), there are still 2 elements unaligned (compulsory conflicts). Likewise in 3D, Fig.3.10show that padding a single dimension is not sufficient to improve the alignment.

A *multi-dimensional padding* consists in adding elements at the end of each dimension. The number of elements added is constant per dimension, but may vary from one dimension to another. This flexibility allows more elements in a stencil pattern to be aligned. To define the number of elements p_k to add in each dimension k, we describe the relation between the value of the padding and the stride separating two consecutive elements of the array. For any dimension k, the stride S_k separating two consecutive elements in the k^{th} dimension has to take into account the size taken by all elements in dimensions h, $h < k$, and any padding on these dimensions. This provides a recurrent definition of the stride for dimension

k : it is equal to the stride of the $k-1^{th}$ dimension times the number of elements in this dimension, plus the padded elements for dimension $k-1$. Formally, the stride is defined by :

$$S_k = S_{k-1}V_{k-1} + p_{k-1}.$$

In a matrix form, this defines a relation between paddings and strides :

$$\begin{pmatrix} p_1 \\ \vdots \\ p_{n-1} \end{pmatrix} \equiv_l \begin{pmatrix} -V_1 & 1 & & & \\ & -V_2 & 1 & & \\ & & \ddots & \ddots & \\ & & & -V_{n-1} & 1 \end{pmatrix} \begin{pmatrix} S_1 \\ \vdots \\ S_n \end{pmatrix}, \tag{3.17}$$

where only non-null elements are represented in the matrix. As expected, the value of each padding is bounded by the length of the SIMD vector (due to the relation $\equiv_l$). Besides, as soon as strides of two successive dimensions are multiple of the vector length (maybe thanks to padding of these dimensions), Equation 3.17 shows that there is no need for padding in outer dimensions.

3.5.3 Finding a Multi-dimensional padding

Equations (3.16) and (3.17) provide the constraints on the padding so as to obtain potentially an alignment for all elements of a stencil pattern. Since padding consists in adding extra, useless elements, the total count of such elements should be minimized in order to reduce the impact of this method.

Besides, for some stencil patterns, even a multi-dimensional padding cannot ensure that all elements are aligned. For each unaligned element, either a shuffle or a misaligned load will be generated. To count these necessary shuffles, according to the padding chosen, we reformulate Equation (3.16) so that a solution can always be found, using slack variables w_k :

$$\begin{pmatrix} d_{11} & \dots & d_{1n} \\ \vdots & \ddots & \\ d_{m1} & & d_{mn} \end{pmatrix} \cdot \begin{pmatrix} S_1 \\ \vdots \\ S_n \end{pmatrix} \equiv_l \begin{pmatrix} c \\ \vdots \\ c \end{pmatrix} + \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix} \tag{3.18}$$

Here, w_h stands for an additional shift on the element h of the stencil pattern, corresponding to an alignment change. As for any shift, $0 \geq w_h < l$. In order to count and minimize the number of elements for which an alignment change is necessary, we add the constraint for any element :

$$0 \leq w_h < (1-u_h)M \tag{3.19}$$

with M a big constant. u_h is a $0-1$ variable equal to 1 whenever a shuffle or an unaligned load is needed.

An objective function to minimize for the multi-padding problem, combining the minimization of the number of unaligned loads/shuffles and the memory consumption due to padding can be defined as :

$$\min(\sum_{h=1}^{m} u_h + \sum_{k=1}^{n} p_k) \tag{3.20}$$

A different objective function may be formulated, depending on the architecture and on the cost associated to memory consumption and shuffles. The complexity and the number of equations to consider force the use of a linear solver.

For the example in Fig.3.11, the multidimensional padding (1 in each dimension) enable to align all elements of the stencil pattern. There is no need for unaligned access or shuffle.

3.5.4 Formulation of Multi-dimensional Multi-Padding (MDMP)

In order to further reduce the number of shuffles or unaligned accesses to load a stencil pattern, we describe now an extension of the multi-dimensional padding. The principle can be easily explained for a 2D array.

For a 2D array, each row can be padded by a number of elements. Instead of padding each row with the same number of elements, each row is now padded with a number of elements that can differ from row to row. In order to keep the code generation with this padding manageable, the padding is cyclic : every T rows, the sequence of padding values starts over again. This technique can be generalized to any number of dimensions. The Jacobi 2D in Fig.3.7 shows that by padding even rows with one element, and odd rows with 3 elements, all elements of the stencil pattern can be aligned but one. Note that the remaining unaligned access is a compulsory alignment conflict, since the two elements E and G are in the same row. The padding proposed minimized the number of unaligned accesses.

We consider here some periodic functions $S_k(j)$, defining the strides separating consecutive elements of A in the dimension k. The address of an element of the array A is defined by :

$$\&A[i_1, \ldots, i_n] = A + \sum_{k=1}^{N} \sum_{j=0}^{i_k} S_k(j) \tag{3.21}$$

If T is the period of the functions S_k, then S_k is characterized entirely by the set of values $\{S_k(0), \ldots, S_k(T-1)\}$. This enable the following simplification of formula (3.21) :

$$\&A[i_1, \ldots, i_n] = A + \sum_{k=1}^{N} \sum_{j=0}^{T} \lfloor \frac{i_k - j}{T} \rfloor S_k(j).$$

Considering the elements of the stencil for the computation in $(i_1, \ldots, i_n)$, they are all aligned if and only if :

$$\exists c, \forall h, \sum_{k=1}^{n} \sum_{j=0}^{T} \lfloor \frac{i_k - j + d_{hk}}{T} \rfloor S_k(j) \equiv_l c$$

for some constant c. Let us denote $s_{jkh}(i) = \lfloor \frac{i-j+d_{hk}}{T} \rfloor - \lfloor \frac{i}{T} \rfloor$. The value of $s_{jhk}(i)$ can only take two values at most when i changes, and $s_{jhk}(i)$ is a periodic function of period (at most) T. The condition for alignment then becomes :

$$\exists c, \forall h, \sum_{k=1}^{n} \sum_{j=0}^{T} (\lfloor \frac{i_k}{T} \rfloor + s_{jkh}(i_k)) S_k(j) \equiv_l c$$

Writing this constraint in matrix form :

$$\exists c, \begin{pmatrix} s_{111}(i_1) & \ldots & s_{1nT}(i_n) \\ \vdots & \ddots & \\ s_{m11}(i_1) & & s_{mnT}(i_n) \end{pmatrix} . \begin{pmatrix} S_1(0) \\ \vdots \\ S_1(T-1) \\ \vdots \\ S_n(T-1) \end{pmatrix} \equiv_l \begin{pmatrix} c \\ \vdots \\ c \end{pmatrix} \tag{3.22}$$

This constraint of alignment is similar to Equation (3.16). The periodicity of the strides lead to consider a larger stencil pattern (matrix has size $m \times nT$). Similarly to Equation (3.17), strides between elements and padding are connected through the equation :

$$\forall k > 1, \forall h, S_k(h) = \sum_{j=0}^{T} \lfloor \frac{V_{k-1} - j}{T} \rfloor S_{k-1}(j) + p_{k-1}(h). \tag{3.23}$$

Padding for Bank Conflicts

For multithreaded stencil codes, we assume that the parallelization is such that one or several of the dimension of the stencil correspond to parallel loops. This means for instance that some indices i_k can be written as $i_k = b.t + ii_k$ where t is a thread number, ii_k an index enumerating the block in this dimension allocated to a given thread. Other partitioning of an array dimension can be considered, as long as the partitioning correspond to an affine transformation. However, the parallelization is an input for the padding and the exact distribution of iterations among threads has to be known before padding.

To minimize bank conflicts, each elements accessed simultaneously by each thread must hit a different memory bank. Hence, for each couple of threads, the address of their first element must not be on the same alignment. For a given address m, the number b of the memory bank corresponding to it corresponds to some contiguous bits of m : $b = m/B \mod NB$ where B and NB depend on the architecture. Thus, following the representation of the addresses in previous section, we can express that each thread accesses different memory banks : if $b(t_1, ii)$ is the memory bank accessed by thread 1 through a memory access, and $b(t_2, ii)$ is the memory bank corresponding to the same access in the block for thread 2, then the following equation corresponds to the constraint :

$$b(t1, ii) - b(t2, ii) \equiv_{NB} db,$$

with NB the number of banks and $db \geq 1$.

As for the padding for vectors, slack variables following the same constraint as in equation 3.19 can be added in order to ensure the existence of a solution. Hence, we will end up with the same objective function as in Eq.3.20, trying to minimize first the number of slack variables representing shuffles, and then the padding values. Having the same objective function, we can easily merge all the constraints (for vector alignment and bank conflicts) to answer all the requirements in the same linear system.

Resolution and code generation with MDMP

Equations (3.22) and (3.23) define the constraints on the periodic constraint. Provided the period T is given and constant, equations are similar to those considered in 3.5.3. The number of equations has increased due to the periodic values and due to the fact that the coefficients of the matrix in formula 3.22 are also periodic (in $(i_1, \ldots, i_n)$).

The formulation of the function to minimize, depending on the number of shuffles, is similar to the one presented in section 3.5.3.

For code generation, partial loop unrolling by a factor T enables an easier address computation, taking into account the periodic padding.

3.6 Code generation

Code generation for stencils on CPUs or GPUs consists in two phases :

1. **Memory allocation and data transfer** : The resolution of the system of constraints relative to multi-padding provides the necessary amount of memory to allocate (or re-allocate) for effectively pad the different dimensions of the data structures. Since the data layout transformation consists in translating data in memory, any required copy (for instance for GPU) is easily generated.
2. **Instruction generation** : Once the data layout is computed and memory is allocated, the stencil code is generated. Memory instructions are generated taking into account the new data layout. Aligned and unaligned instructions can be necessary, and the two different ways to generate these instructions are presented in Section 3.6.2.

3.6.1 Efficient data layout

Algorithm 2 describes the different steps to remove a maximum number of inefficient memory accesses (bank conflicts or unaligned aligned data accesses).

Algorithm 1 Efficient Multi-Padding and Memory Management

```
1: Find Multidimensional Multi-padding to remove bank conflicts while aligning data for vectorization.
2: Memory Allocation
3: Data transfer, and Copy from High Latency Memory (HLM) to Low Latency Memory (LLM)
```

Padding for Vectors

Padding for vector is used to align a maximum of required elements in the computation pattern. Aligned elements will be accessed through more efficient memory transactions. The complete procedure is describe in Section 3.5.1. The constraints are the same for CPUs and GPUs when using built-in vector types in CUDA.

However, on GPUs, hardware computations use only on element at a time. If one uses its own vector structure instead of CUDA built-in types, no specific alignment is required, and this step may be skipped, remaining only to avoid bank conflicts between the threads.

Thread parallelism

As explained in Section 3.2.1, simultaneous memory accesses from concurrent threads should target different memory banks. Aligning the first data accessed by each thread on a different value is sufficient to ensure such repartition, minimizing bank conflicts. We consider that each thread will run the same computation.

CPU threads : On a multicore CPU, we consider that the original iteration domain is sliced for each thread to access an independent block of the array. A block limits are only on a dimension boundaries, and a block can not begin or end in the middle of a line.

Avoiding banks conflicts consists in finding an additional multipadding to ensure that the first memory access performed by each thread will not collide on the same bank. The additional cells required are added at the end of array blocks, which is possible since a block boundary matches an hyperplane boundary.

GPU thread warps : GPU threads can be considered as CPU threads, and the GPU code can be designed for each thread to access an independent block of data. However, warps often access contiguous data in the shared memory, since there is no time consuming memory manager to trigger. To generate such a GPU code, one has to know if the whole warp will access contiguous data (in that case there is nothing to do), or if slices of warp will be spread across several lines/blocks. For example, a warp being a set of 32 threads, the code can be designed for the warp to access 32 contiguous data, or for four quarter of warps to access 8 contiguous data on different lines. These slices will then be considered as meta-vectors (4 in our example), which should not have any memory banks in common . The multipadding method will be applied with these meta-vectors, using their specific constraints.

Memory Allocation

Once all the necessary paddings found, one must combine them to know the total amount of memory to allocate.

On GPU, memory will be allocated to the shared memory of a chip, according to the padding found in the first step of the algorithm. Then, in a next step, necessary data will be transfer from global memory to the shared memory, closer and easier to access efficiently.

On CPU, a global memory allocation will be made. This global allocation will include the padding for vectorization, and the padding for threads at end of logical thread block.

From HLM to LLM

Data transfer consists in putting all needed data for computation on the correct device memory. It will be a copy to the global memory if GPU is considered.

In order to reduce the cost of memory transactions, or to simplify memory accesses pattern, a first copy from the High Latency Memory, in which the data are originally stored, to a Low Latency Memory can be useful.

On GPU, data will be transfered from the global memory to the shared memory. Global memory has a very restrictive access, since all threads of a warp should access contiguous data with a specific alignment to perform efficient memory movement (coalesced loads). On the other hand, threads can load data from shared memory very easily, and the concurrent memory transactions can be spread across the entire array without penalties (if bank conflicts are avoided).

CPU memory hierarchy is simpler since the wherever the data are, they are fetched to the nearest memory to the computing unit each time they are read. The only needed data transfer is from the original array to the new allocated region, modifying the data layout.

3.6.2 SIMD code generation

For CPU code generation, we rely on state-of-the art compiler vectorization techniques to generate SIMD code. Two approaches are taken :

- Vector extensions proposed by compilers (`icc` and `gcc`, with attribute directives)
- Intrinsic functions

We favor in the code generation the first approach, whenever aligned elements are accessed. The advantage of this method is that instruction selection is actually performed by the compiler, depending on the target architecture and target vector ISA. However, this method is limited to basic algebra operators and all memory accesses to a vector have to be all aligned to be efficient. These limitations narrow the number of stencil patterns that can be handled, and only a Jacobi pattern in double precision can be efficiently produce. For other cases, we favor accesses to unaligned vector elements, aligned accesses and shuffles are used. Besides, SSE2 intrinsic functions are then used.

3.7 Related Works

For generalist multicore processors, there has been considerable interest on vectorization issues. Most papers focus on loop transformations to improve the vectorization quality, playing with data locality to decrease the amount of unaligned loads, or shuffles ([76, 100, 78, 46, 77, 79, 110, 24, 42]). Papers on stencil computations follow the same trend, with most papers focusing on improving data locality through blocking ([57, 76, 34, 106, 99, 73, 40, 41]), tiling ([89, 60, 69]) or skewing ([117]) to improve single and multi-core performance. Unfortunately, all these works do not directly consider alignment issues, focusing only on data locality and register reuse.

For GPUs, generating efficient code able to close the gap with the peak performance, is quite challenging [66]. Several auto-tuning models applying regardless on CPUs and GPUs have been offered to ease programming on multiple target architectures ([12, 73, 106]), without considering the common alignment issues.

Some recent works focus on data layout transformations ([70]) to improve aligned vectorization for stencil codes. The work of Henretty *et al.* [52] focuses on a data layout transformation for stencil vectorization. Data reorganization, through dimension lift en transpose, remove all non-aligned loads. Contrary to multipadding, they do not consider multithreaded execution. Originally compact data are spread across the whole array, and the memory sharing between threads will probably induce performance slowdown.

The Pochoir project [122, 95] provides a compiler and a runtime system for implementing stencil computations on multicore processors, with the user specifying its computing kernel with specific language embedded in C++. Using the Pochoir compiler, the describe stencil computation is optimized, only considering cache issues. Two papers [35, 56] offer an auto-tuning framework to optimize stencil computations on multicore architectures including GPU. These works focus on blocking for the different memory levels, and no alignment issue is addressed. Performance figures bring some comparison basis that will be used in section 3.8. Since no data blocking is realized in our method, the two approaches can be combined to obtain better performance.

3.8 Benchmark results

To evaluate our method, we run several tests of stencil computations (Jacobi and Laplacian), on several intel architectures along with a GPU environment, and all tests have been done with icc 12.1.0 and gcc 4.6.1 compilers.

The multipadding method was also applied on a 9-points 2D stencil, and a 27-points 3D stencil using Moore's neighbourhood, and on a diamond-shape pattern in 2D using a Von Neumann neigbhourhood of range 2. Performance results are not presented since the best possible padding for the 9-points and 27-points stencil is no padding at all. For the diamond-shape pattern, the best possible padding is a simple padding, and no other results on such pattern are available for comparison.

Performance results in section 3.8.2 are for computations in simple precision, as it is more difficult to align data when vectorizing such codes (four possible positions in vectors). Tested codes have been generated with intrinsics performing aligned loads on correctly aligned data, and unaligned loads otherwise. Graphs bars labelled "No pad", used as basis for comparison, are for already vectorized versions of the stencil computations on which no layout modification has been applied.

Performance results in section 3.8.3 are for computations in double precision, since the results we use for comparison have been presented only in double precision. Tested codes have been generated with vector extension, and loads are automatically managed by the compiler. Note that for all experiments, no blocking is realized to increase data locality and reuse.

3.8.1 Target Architectures

We evaluated our generating approach minimizing alignment conflicts on several architectures described below. We used three Intel CPUs, with one Xeon architecture, one Nehalem, and a more recent Nehalem with the Westmere Gulftown processor. In addition to the CPUs, tests have been run on a GPU NVIDIA Quadro card, plugged to the Nehalem machine.

Intel Xeon Clovertown E5345

The Clovertown architecture is Intel's first quad-core processor, consisting in two dual-core Woodcrest paired onto a multi-chip module. Each core runs at 2.33 GHz. Supporting SSE3 SIMD instructions, each core can fetch and decode four instructions per cycle, for a peak performance of 9.32 GFlop/s per core in double precision. Each core includes a 32KB L1 cache, and each dual-core chip has a shared 4MB L2 cache. Our machine presents two sockets, for a total of 8 available cores. Each socket has access to a 333MHz quad-pumped front side bus (FSB) delivering a raw bandwidth of 10.66 GB/s.

Intel Nehalem Gainestown X5550 and NVIDIA CUDA Quadro 5800 FX

The Gainestown architecture is a quad-core processor with each core running at 2.66 GHz. Supporting SSE4 SIMD instructions, each core can fetch and decode four instructions per cycle, for a peak performance of 10.64 GFlop/s per core in double precision. Each core includes a 32KB L1 Data cache and a 32KB L1 instruction cache, and a 256 KB L2 cache/core. Each cores can access a 8MB L3 shared cache. Our machine presents two sockets, for a total of 8 available cores. Each socket has access to a FSB delivering a raw bandwidth of 12.8 GB/s.

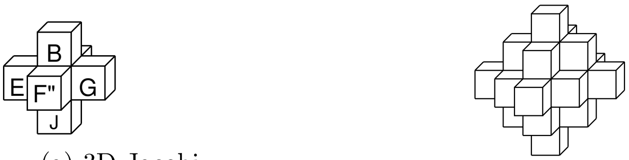
Our Gainestown machine is equipped with a NVIDIA Quadro 5800 FX card, running at 1.30 GHz and with 4GB of memory. This card was used to perform GPU tests in section 3.8.

Intel Westmere Gulftown X5650

The Westmere architecture is the evolution of the Nehalem architecture. The processor used for the benchmarks is a six-core based processor. It runs at the same speed, for the same peak performance of 10.64 GFlop/s. The L3 cache shared between all six cores has a size of 12MB. Our machine contains four sockets, for a total of 24 available cores.

3.8.2 Overall improvement

In order to validate our approach, the first part of our experimentation testbed consists in trying several padding values, for simple padding and multipadding, to see if the best padding(s) return by the methods discussed in section 3.5.1 were indeed the best one(s).

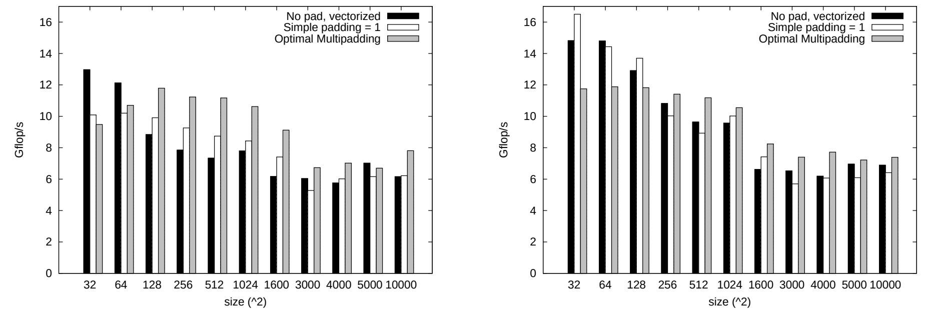


(a) 3D Jacobi

(b) 3D Von-Neumann

FIGURE 3.12 – Shapes of the studied stencil. (a) Shape of a 3D jacobi stencil. (b) Shape of the convex envelope of a 3D Von-Neumann neighbourhood of range 2.

In Fig. 3.13 are shown results for a 4-point 2D jacobi stencil computation, running on only one thread. The multipadding vesion is compared with versions with no padding, and the usual simple padding. Fig. 3.13**(a)** presents results when using the gcc compiler, on different square sizes (from L1 to memory). Except for data in L1, the multipadding outperforms the other versions of the Jacobi 2D stencil, with a performance improvement up to 52% for a size of 512x512, and a gain of 28% compared to the simple padded version on the same size. The same experiment as in Fig.3.13**(a)** is displayed in Fig. 3.13**(b)**, only this time with the use of icc compiler. As the compiler use more aggressive and guided optimization, the produced code without padding already performs well. The multipadding method still gives better performance when in upper hierarchy memory (L3 begins at size 256), up to a x1.24 speedup for a 1600x1600 array against the original version, and a x1.11 speedup compared to the padded version for the same size.



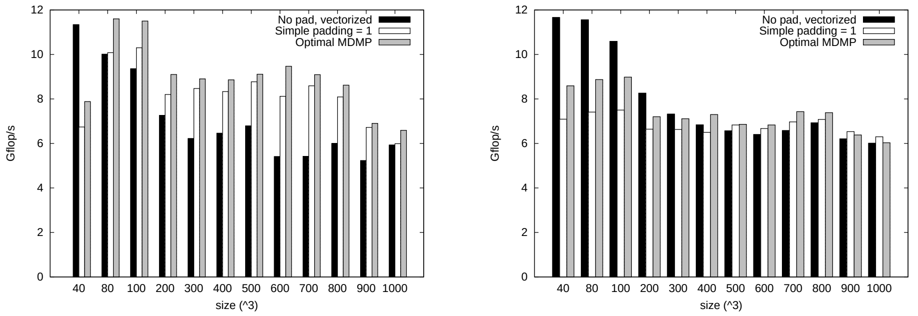
(a) Results of Jacobi 2D with gcc

(b) Results of Jacobi 2D with icc

FIGURE 3.13 – Performance of 2D Jacobi stencil on Westmere for different square sizes.

In Fig. 3.14 are shown results for a 6-point 3D jacobi stencil computation (Fig.3.12), running on only one thread. The multipadding vesion is once again compraed with versions with no padding, and the usual simple padding. Fig. 3.13**(a)** presents results when using the gcc compiler, on different square sizes (from L1 to memory). Except for the smallest tested size, the multipadding outperforms the other versions of the Jacobi 2D stencil, with a performance improvement up to 61% for a size of 600^3, and a gain of 22% compared to the simple padded version on the same size. The same experiment as in Fig. 3.13**(b)** displays the results for the same experiments using the icc compiler. As for 2D Jacobi benchmarks, the produced code without padding already performs well. The multipadding method still performs better than the other versions when accessing memory, but with less significant improvement (around 5% speed-up).

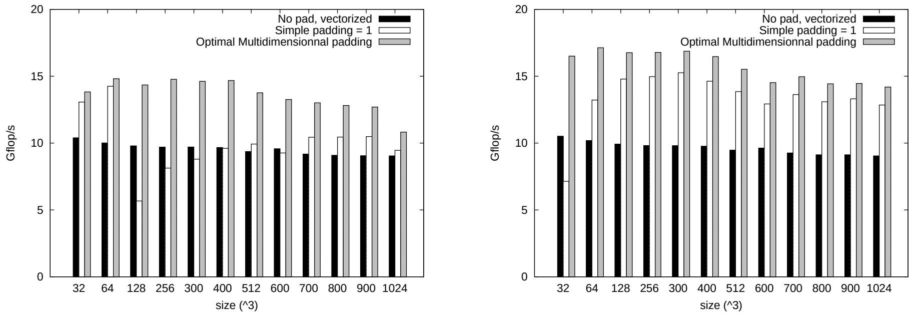
In Fig.3.15 are presented results for a stencil covering the convex envelope of a



(a) Results of Jacobi 2D with gcc (b) Results of Jacobi 2D with icc

Figure 3.14 – Performance of 2D Jacobi stencil on Westmere for different square sizes.

3D Von-Neumann neighbourhood of range 2 . The shape of this stencil pattern is displayed in Fig.3.12**(b)**. As before, the multipadding version is compared with version with no padding, and the usual simple padding. Here, only a multidimensional padding is sufficient, i.e. each dimension has been modified with only one padding value. This multipadding allows to align 10 elements in the stencil, instead of only 6 elements for the version with no padding and with the simple padding. Figures 3.15**(a)** and 3.15**(b)** presents results when using respectively gcc and icc compilers. For each set of benchmarks, the multipadding version always outperforms the other version, up to a performance improvement of 52% compared to the original version for a size of 256^3 with gcc, and 72% for a size of 300^3 with icc.

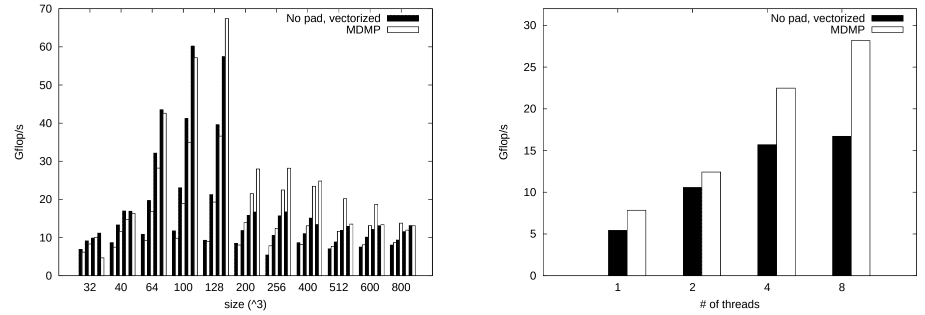


(a) Results of Von-Neumann 3D with gcc (b) Results of Von-Neumann 3D with icc

Figure 3.15 – Performance of stencil based on 3D Von-Neumann neighbourhood of range 2 on Intel's Sandy Bridge for different square sizes.

Some multi-threaded performances are displayed in Fig.3.16 for a 6-point 3D Jacobi compiled with icc, with simple precision data. Each thread is pinned to a different core, and no hyperthreading is considered. The results are presented in Fig.3.16**(a)** for several sizes. On each cluster of eight bars, the two first ones stand for only 1 thread, the two following bars show results for 2 threads, and so on up to 8 threads. The white bars always represent the original vectorized code, and the black bars our multipadded code version. As we observe the same behaviour with 1 and 2 threads as in Fig.3.14**(d)**, significant performance gains are achieved when hitting the RAM, up to 70% with 4 threads, and 85% with 8 threads, compared to the original vectorized version with the same amount of threads.

A close up for results on a 256^3 matrix is presented in Fig.3.16**(b)**, since data are in memory, and this size is the one presented in most of stencil related papers. The multi-dimensional multipadding version is compared to the original version with no padding. The MDMP version outperforms the original version each time, with a 45% improvement for one thread, and up to a 69% improvement for eight threads.



(a) Results of multithreaded Jacobi 3D for several sizes

(b) Close-up on results of multithreaded Jacobi 3D of size 256^3

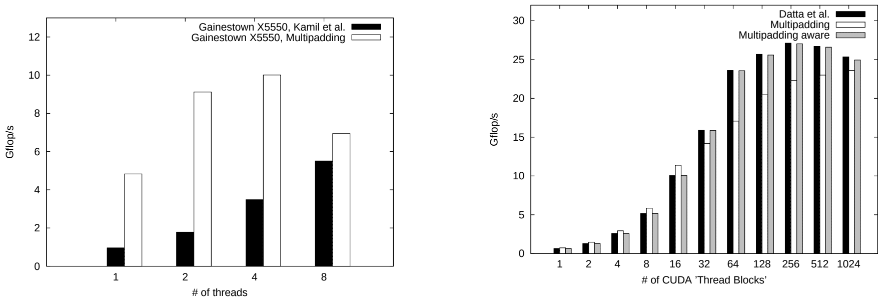
FIGURE 3.16 – Performance of multithreaded Jacobi 3D stencil on Westmere architecture, using one thread per core.

3.8.3 Comparing with related works

In this section, results are in double precision and compared with results presented in related works, on a 7-point 3D Laplacian stencil computation. In Fig.3.17**(a)** we reported results shown in Kamil *et al.* [56], and presented our own performance figures obtained by applying a multipadding method (here, thread alignment and a multi-dimensional padding) for the 7-point 256^3 sized Laplacian stencil. Benchmarks for the Nehalem architecture (Fig.3.17**(a)**) have been realized on the same Gainestown architecture.

On the Clovertown, the poor scalability to more than 2 cores kills the performance gain our method showed for 1 and 2 cores (up to 40% of improvement). On the Nehalem, our method confirm its great results when in memory and for multiple cores. MDMP performances are more than 4x better with 2 and 4 cores than Kamil *et al.*

For GPU figure (Fig.3.17**(b)**), we were able to run the code from Datta *et al.* [35] on our machine, allowing to compare directly the results on the same basis. As our code generator does not produce CUDA code directly, only the code for the threads have been produced by the generator. All data transfers, and memory and thread allocations, are performed by hand. The white bars (labelled "Multipadding") give results when the code is generated taking into account CUDA programming guide's [31] recommendation : one stencil update per thread. However, in Datta *et al.*, it is stated to compute 4 stencil updates per thread. A code modification have been made subsequently, and performance results are displayed in grey bars (labelled "Multipadding aware"). As our first version can not withstand the comparison, the "aware" version of the CUDA code successfully sustains around the same performance as Datta *et al.*, until beginning to take off for a great number of CUDA threads blocks (>128).



(a) Results on Nehalem architecture

(b) Results on GPU Quadro 5800 FX architecture.

FIGURE 3.17 – Performance of a 256^3 Laplacian stencil for multipadding method (a) Kamil *et al.* [56] on Gainestown, and (b) for Datta *et al.* [35] on GPU. Multipadding results are obtained with one computation/thread (CUDA programming guide recommendation). Multipadding aware results are obtained with 4 computations/thread (Datta *et al.* recommendation).

3.9 Conclusion

Stencil computations correspond to codes at the heart of a large class of applications. Due to the nature of stencils, alignment, memory management and vectorization are the key to performance, for GPU and CPU architectures alike. The contributions of this chapter are :

- A data-layout transformation so as to handle CPU and GPU alignment issues. This transformation called multi-dimensional multipadding introduces new elements in multi-dimensional structures and generalizes the usual padding transformation.
- A compact code generation for stencils. The data-layout transformation only requires to unroll partially loops and translate some indices of data structures.

The model based on integer linear programming is broad enough to handle the multicore CPU and GPU cases. Other alignment conflicts problems can be easily added, providing the correct objective function. Experimental results on multiple targets demonstrate the validity of the approach on several stencil kernels with x1.52 speed-up on one core, and up to x1.69 speed-up on multicore. Performance obtained compared well with previous work on stencils, outperforming previous performance obtained on multicores and reaching similar level of performance on GPU.

Future works

Since our experiments were realized without any blocking for cache locality, a first step would be to measure performance on codes with cache blocking and/or register blocking. All experiments display speedups for data in L3 cache. Blocking for this cache level should produces better results than the presented ones. Whereas for L2 cache, only some of our experiments display a speedup, and it would be interesting to observe how blocking and multipadding interact on lower cache level.

Also, one of our future endeavour is to produce results for a wider range of stencil pattern. Using Von Neumann's neighbourhood, and some derivatives, of several orders, we can consider multiple stencil patterns used in practice, and interesting for our multipadding method. Patterns other than those using Moore's and Von Neumann's neighbourhoods of the first order are not well represented in papers. Besides observing the general speedup that multipadding can bring on these stencils, it can be useful for us to provide a large set of experimental results, offering the community other comparison points.

When we produced the code for the 3D Jacobi stencil, it appeared that the data layout obtained when solving the linear system was not the best one achievable on this pattern. The constraints we use to generate the system do not allow to find this best data layout. However, some light modifications on those constraints will achieve the best layout. Instead of having the same inner multipadding for each hyperplane in a dimension, we can allow for several hyperplanes to have a different inner multipadding, with the layout of these hyperplanes repeating regularly. Proposing other models, more or less complex, is a continuation of the multipadding method.

Finally, more generally on stencil computation, I am curious about trying some optimizations which were not yet studied a lot in papers. For example, time unrolling, hence doing several update steps at the same time, can shift the bottleneck of the code from memory to computation. On the other hand, applying a loop reversal between two update loops will allow to reuse the data in L1 and L2 caches, even if the whole stencil is blocked for L3 cache, or runs in memory. For this last point, the speedup may not be that important, but even a few percents gain can represent a good and concrete time, energy, and/or money saving on large simulation programs using stencil codes.

Chapitre 4

Towards more automatic parallelization

In this chapter are discussed the two ways of introducing parallelism in a task graph : either each task is sequential and multiple tasks can run concurrently, or each task has parallel sections and only one task can run simultaneously. Two distinct works realized with different partners explore each possibility

This chapter is complementary with the main focus of this thesis described in the previous chapters :

– The CPC framework : since no parallelization was considered in Chapter 2, the methods described in this chapter can be used to insert parallelization in the framework. The CPC framework handling a task graph composed with several codelets, the user has the choice between scheduling the tasks in parallel, or introducing the parallelization as a possible transformation to try when optimizing each codelet.
– Multi-Padding : as the different paddings are chosen to avoid bank conflicts between simultaneous threads, some schedule between the different parallel task is supposed to be known when applying the method. This schedule coming from any sources, we can easily consider to couple one of the method presented in this section, scheduling a task graph, with the Multi-Padding method, finding the best data layout corresponding to the returned schedule.

The first part of this chapter describes how to use C++ Expression Templates to directly generate tasks realizing parallel vector operations. Expression Templates allow to consider any set of operations as an expression being carried across function calls, until its complete evaluation is actually needed. Generally, evaluation is done through usual C++ implementation of operators. However, it is possible to consider the set of operations in straight C code in a first step, then evaluate through our CPC framework described in Chapter 2 the best implementation to consider when evaluating such expressions.

The second work reported in this section concerns parallel tasks scheduling in an heterogeneous environment. This work interacts greatly with the main focus of this thesis. When the CPC framework is applied on the hot-spot of a program, we end up with a bunch of constant performance codelets, either independent or relative to each other. Adding a dependence graphs between the codelets, one can consider them as tasks which can be scheduled with MSFT scheduling approach described further. Furthermore, with the work on the stencil codes in Chapter 3 (more specifically in section 3.6), we developed a method to generate stencil codes efficiently for CPU and GPU. On an heterogeneous machine composed with multiple cores of each type, one will be able to decompose a large stencil is several part running efficiently on each core with this method, then schedule each part with MSFT scheduling algorithm.

These works are presented respectively in the POOSC'09 workshop [84] and in another paper currently revised following the result of a TPDS-2011 conference submission.

4.1 Sequential execution of tasks with embedded parallelism

This section proposes a simplified C++ implementation of a linear algebra vector class that allows composing compact and abstract vector expressions such as :

$$\mathtt{X} = \mathtt{a} * \mathtt{Y} + \mathtt{b} * (\mathtt{Z} + \mathtt{W}); \tag{4.1}$$

This implementation is based on the Expression Template mechanism (ET) introduced by Veldhuizen [104] and Vandevoorde [103]. ET allows avoiding temporary vectors and the performances of abstract expressions like (4.1) compete with the ones of the corresponding low-level (loop-based) basic implementations such as :

$$\mathtt{for}\ (\mathtt{i} = \mathtt{0};\ \mathtt{i} < \mathtt{N};\ \mathtt{i{+}{+}})\ \mathtt{X[i]} = \mathtt{a} * \mathtt{Y[i]} + \mathtt{b} * (\mathtt{Z[i]} + \mathtt{W[i]}); \tag{4.2}$$

In a previous work [82], L. Plagne and F. Hülsemann illustrated the gain to be obtained from mixing the ET based vector class with the ATLAS [113] implementation of the procedural BLAS library. The high performance ATLAS implementation of the vector copy operation, which relies partly on low level assembly language, is used as a kernel for the vector ET evaluation.

In this following work, we present two multi-threaded implementations of our ET based vector class based on Intel's Threading Building Blocks and on OpenMP respectively, thus generating directly tasks with parallelism. The ubiquitous presence of multicore architectures has rekindled the interest in shared memory parallel programming models. While the computational power of a chip scales up almost ideally with the number of cores, this is not the case for the memory access bandwidth. Hence the fraction of the so-called *memory bound* applications, which feature performances that are limited by the memory access bandwidth of target architecture, increases with the mean number of cores included in micro-processors.

This section is organized as follows : Subsection 4.1.2 presents the considered vector operations and the large vector operations as typical memory bound problems. Subsection 4.1.3 gives a short description of our C++ vector class implementation based on ET. Performance measurements show that this implementation avoids abstraction penalties. Subsection 4.1.4 presents our enhanced ET vector class relying on the ATLAS dcopy kernel. Performance measurements are carried out on three different architectures. Subsection 4.1.5 presents two parallel implementations of our ET vector class and the corresponding results.

In Section 4.2, the second work focusing on a new approach for parallel tasks scheduling on heterogeneous platform will be described. Section 4.2.5 provides an overall conclusion regarding contributions of each works and their interactions with the previous chapters.

4.1.1 Target Architectures Description

The benchmarks have been carried out on different x86_64 target architectures. Table 4.1 provides a short description of the main features of theses machines. In the following, the performance curves will be named after these three architectures.

Table 4.1 – Description of the three target architectures for performance measurements

Processor	# cores	frequency	RAM	Compiler
Intel Xeon E5570 Nehalem	2×4	2.9 GHz	18 GB	g++ 4.3.3
Intel Xeon X7310 Tigerton	4×4	1.6 GHz	48 GB	g++ 4.3.3
Intel Xeon E5410 Harperton	2×4	2.3 GHz	8 GB	g++ 4.3.3
AMD Opteron	2×4	1.9 Ghz	4 GB	g++ 4.3.0

Most of the time, the different targets exhibit the same kind of performance behavior. In this case, we will report only the Xeon E5410 curves. The performance measurements are carried out with the tools developed by the BTL++ project [83].

4.1.2 Vector Operations

Vector expressions like (4.1) exhibit a small arithmetic intensity and are strongly *memory bound*. Hence it might be surprising that multi-threaded implementation accelerates these tasks significantly on shared memory multi-core machines. Nevertheless, we have observed a ×2.5 acceleration factor on dual socket quadricore processors compared to our previous implementation. The resulting vector class allows composing abstract vector expressions like (4.1) that achieve a better performance than both loop-based implementations such as (4.2) and off-the-shelf vector libraries such as Blitz++ [105], uBLAS [108] or std : :valarray. This performance reaches a factor of ×2.7 for large vectors.

Considered Vectors

Let us first define the scope of this work and what we refer to as *vector operations*. From the linear algebra point of view, vectors can be defined as *indexed* collections of *numerical* elements of the same type. *Indexed* means that the value of every vector elements can be accessed from a given integer *index* to be chosen in a given *range*. While a wide variety of linear algebra vector types (sparse, multidimensional,...) can be considered, we will focus on simple vector types where real type elements (single or double precision) are stored in basic containers that can be defined and exchanged through the following common programming language : F77, C and C++. Within these languages, the location of a contiguous memory region containing a given number of floating point elements, can be manipulated either as a pointer type (C and C++) or as an array type (F77). These arrays are the main Input/Output types for the Basic Linear Algebra Subroutines (BLAS) API [16].

Performance of Large Vector Operations

Now, let us define we mean by *large* vectors. Fig. 4.1 shows the performance of the vector operation $Y \leftarrow \alpha X + Y$ (axpy) on a Pentium Xeon E5410 using respectively single and double precision floating point elements. Performance is maximal for vector in the range of $[10^2, 10^3]$ elements. From sizes around 10^4, performance decreases and reaches its lowest level when vector sizes exceed 10^5 elements. The reason for this behavior, which is common to all vector operations, is that the performance is mainly driven by the memory access bandwidth. This is true for all

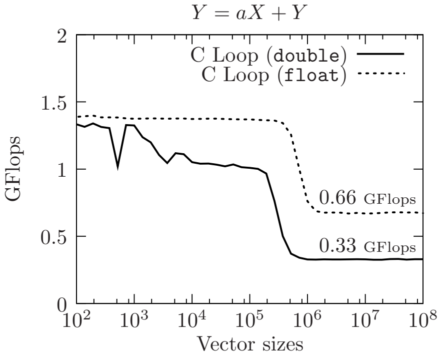


FIGURE 4.1 – Performance results of axpy operations on the Xeon E5410 (gcc 4.3.3).

computations involving a ratio r defined by :

$$r = \frac{\text{number of memory accesses (read+write)}}{\text{number of floating point operations}},$$

that is not small compared to 1. In this case, ($r = 3/2$ for the `axpy` operation), performance depends on memory access bandwidth. Most modern architectures exhibit a memory cache hierarchy with high bandwidth for data access inside the cache (data size $\leq$ a few MBytes) and a much lower bandwidth for data access in the main memory (a few MBytes $\leq$ data size $\leq$ a few GBytes). This explains the low level of performance observed for large vectors that do not fit in the cache hierarchy :

$$\text{Large vectors : vector sizes} \in [10^5, 10^8].$$

Since the double precision `axpy` operation requires twice as much memory bandwidth as the single precision one, it runs naturally 2 times slower for large vectors (0.33 Gflops vs 0.66 Gflops).

4.1.3 Minimal C++ Vector Class

This subsection presents a minimal vector C++ class based on Expression Template mechanism.

Expression Template

ET have been introduced by T. Veldhuizen [104] and D. Vandevoorde [103]. Applied to a vector class, ET allows writing arbitrarily complex vector expressions such as :

```
R=2.0*X+2.0*(Y-Z*2.0);
```

that do not imply temporary vector construction and do not incur any performance penalties compared to the corresponding loop-based implementation :

```
for (int i=0 ; i < N ; i++)
 R[i]=2.0*X[i]+2.0*(Y[i]-Z[i]*2.0);
```

The Curiously Recurring Template Pattern (CRTP)

The proposed ET implementation uses the C++ Curiously Recurring Template Pattern [11, 103] (CRTP) that allows grouping a set of classes in a template-based hierarchy. This hierarchy reflects a common behavior for the class set elements and

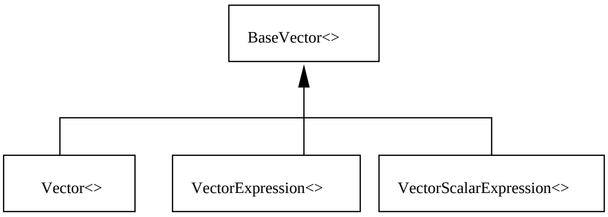


FIGURE 4.2 – BaseVec class hierarchy.

does not involve any virtual functions. As Vandevoorde and Josuttis write, this pattern "*consists of passing a derived class as a template argument to one of its own base classes*" [103] :

```
class Derived : public Base<Derived>{}
```

This pattern is used to gather a set of classes {Derived1, Derived2,. . . } as an ensemble of Base<> classes. In our vector case, let us first define the template base class BaseVec<> :

```
template <class DERIVED>
class BaseVec{
public:
  typedef const DERIVED & CDR;
  inline CDR getCDR( void ) const {
    return static_cast<CDR>(*this);
  }
};
```

The static cast method getCDR() allows extracting the embedded DERIVED object from its BaseVec<> capsule. The DERIVED template class parameter is one of the three following classes : Vec<>, VecExpr<> and VecScalExpr<>.

Fig. 4.2 presents this template inheritance relationship.

VecExpr<> and VecScalExpr<> instances are constructed by arithmetic operators applied to BaseVec<> objects. These two classes store references to the operands. In addition, the VecExpr<> class statically defines the type of operation (+ or -) as a template parameter :

Operators +/- :

BaseVec<L>+BaseVec<R> → *VecExpr<L,Add,R>*

BaseVec<L>-BaseVec<R> → *VecExpr<L,Minus,R>*

Operator * :

*scalar*BaseVec<V>* → *VecScalExpr<V>*

*BaseVec<V>*scalar* → *VecScalExpr<V>*

Operator =

V<T>=BaseVec<T> → *BaseVec<T>[i] evaluation*

Both expression classes define an operator [] that performs the actual evaluation of the expression. Note that this evaluation is not performed at the expression classes construction stage. This is a lazy evaluation process that allows avoiding temporary vectors involved in standard implementations of operators. A more detailed presentation of these classes is given in [82].

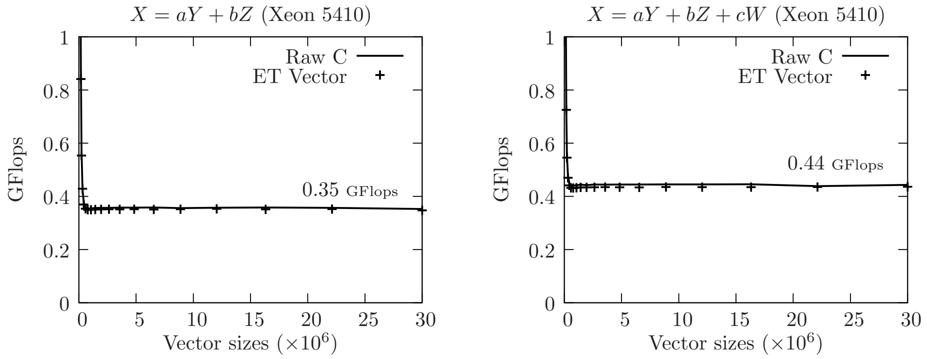


FIGURE 4.3 – Performance results of binary (left) and ternary (right) combinations using the (non-blocked) ET `Vec` class and loop-based implementations (`double` precision vectors ; Xeon 5410 with gcc 4.3.3).

The class `Vec<T>`

The class `Vec<T>` is the main vector class. Its implementation is classical except for the assign operator `=` specialized for `BaseVec<DERIVED>` right hand side :

The template parameter `ELEMENT_TYPE` is either `float` or `double` and the vector elements are stored in a C array of this type (`data_`). The operator `=` evaluates the right operand value that can be the result of an arbitrarily complex expression.

Expression Template Performance

A large variety of vector expressions is handled by this ET vector class implementation. In the following subsections are presented performance measurements carried out for a limited set of vector expressions. This subset should give a fair picture of the general performance level to be expected from the `Vec` implementation.

Considered Set of Vector Operations :

The studied vector operations are linear combinations of vectors :

$$T = \sum_{i=0}^{N_c} a_i S_i,$$

where T is a given target vector, $\{S_i\}$ is a set of source vectors of the same size and $\{a_i\}$ the corresponding set of scalar factors. These combinations can be characterized by the number N_c of involved source vectors :

- $N_C = 1$: unary combinations (part of L1 BLAS API).
- $N_C = 2$: binary combinations.
- $N_C = 3$: ternary combinations.
- ...

Fig. 4.3 shows the performance results of binary and ternary linear combinations implemented via our ET `Vec` class and via direct loop-based C implementations. One can see that the abstract expression of the combinations does not lead to any performance penalties.

4.1.4 Expression Template and ATLAS Based Blocked Evaluation

We combine the previous Expression Template mechanism with a blocked copy technique. The principle is to take advantage of the high performance copy operation provided by the ATLAS library.

The only change in the `Vec` class is a new definition of the operator `=`, where the `BlockAssign` template class implementation is specialized for `double` elements, using the ATLAS copy. Detailed implementation of those modifications are transcribed in our paper [84].

Fig. 4.4 shows the performance improvement enabled by this blocked ET implementation for both binary (+25%) and ternary (+16%) vector combinations.

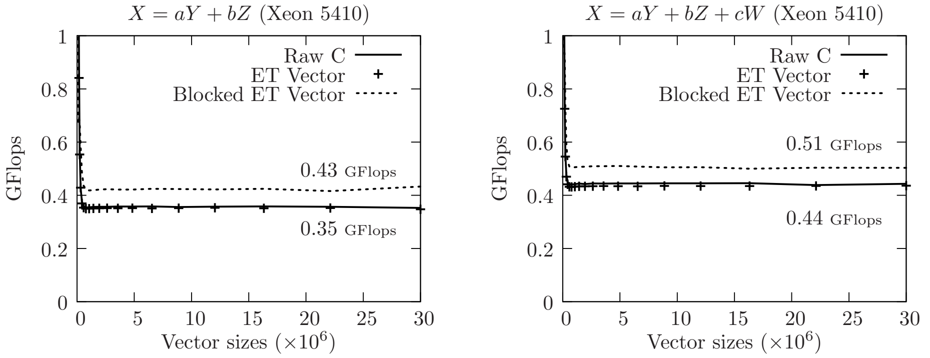


FIGURE 4.4 – Performance results of binary (left) and ternary (right) combinations using the Blocked ET `Vec` class and loop-based implementations (`double` precision vectors ; Xeon 5410 with gcc 4.3.3).

4.1.5 Parallel Expression Template based on Based Blocked Evaluation

In this subsection, we present a parallel implementation of the previous Blocked Expression Template mechanism. The possibility of such an approach is mentioned in the reference [32].

The only change in the `Vec` class is a replacement of the `BlockAssign` template class by the `OpenMPBlockAssign` template class, introducing parallelism inside the copy task. We compared this implementation to another parallel implementation of the `BlockAssign` based on the Intel Threading Building Blocks.

Fig. 4.5 shows the performance improvement that this parallel blocked ET implementation entails for both binary (×2.7) and ternary (×2.6) vector combinations. We have carried out performance comparisons with other available expression template libraries :

- Blitz++ [105],
- uBlas [108],
- std : :valarray.

All these three libraries compared closely to our non-blocked ET implementation for large vectors. Hence the parallel blocking mechanism provides a performance improvement for out of cache linear combinations compared to all vector libraries that we know.

When vector sizes exceed the total size of the cache hierarchy, the performance improvement remains small compared to the number of cores involved in the computation. The main reason is that performance is then driven by the memory band-

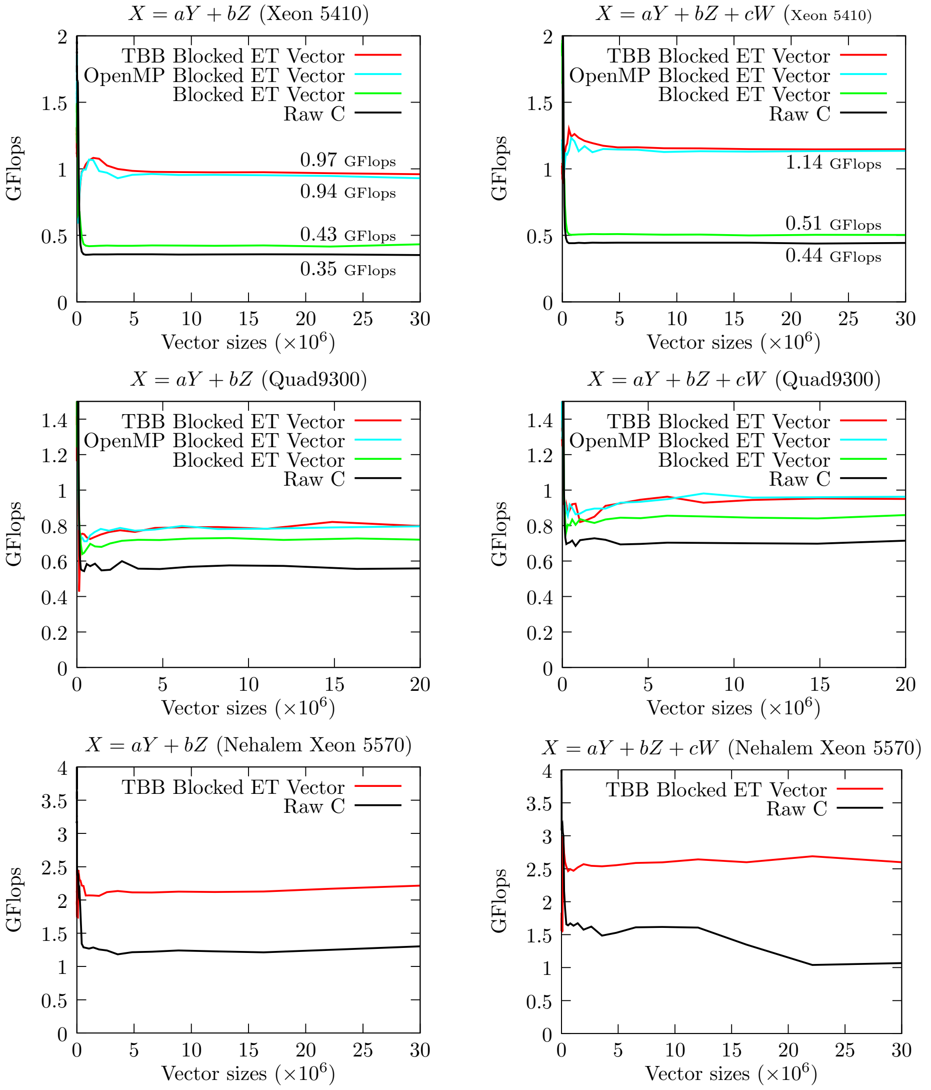


FIGURE 4.5 – Performance results of binary (left) and ternary (right) combinations using the Parallel Blocked ET `Vec` class and loop-based implementations (`double` precision vectors gcc 4.3.3). Top : 8 threads Xeon 5410, Middle : 4 threads Quad9300, Bottom : 8 threads Xeon 5570.

width and other hardware mechanisms such as the size of the in-flight cache requests for each cache.

Fig. 4.6 shows different important results :

- Memory bandwidth usage increases as the number of threads increases. The increase is not linear (between 2 and 4 threads for the Xeon machine for instance), performance only slightly increases. This is due to the fact that cores do not have a uniform memory access. On each chip, four cores compete for the access to memory through the same Front Side Bus. Further more, two cores share the same L2 cache, that can sustain a limited number of in-flight memory requests (cache misses). This could explain for the performance stall in the Xeon machine between 2 and 4 threads.
- Performance of expression template code is comparable to C code. There is no loss of performance, while there is a gain in the expressivity of the

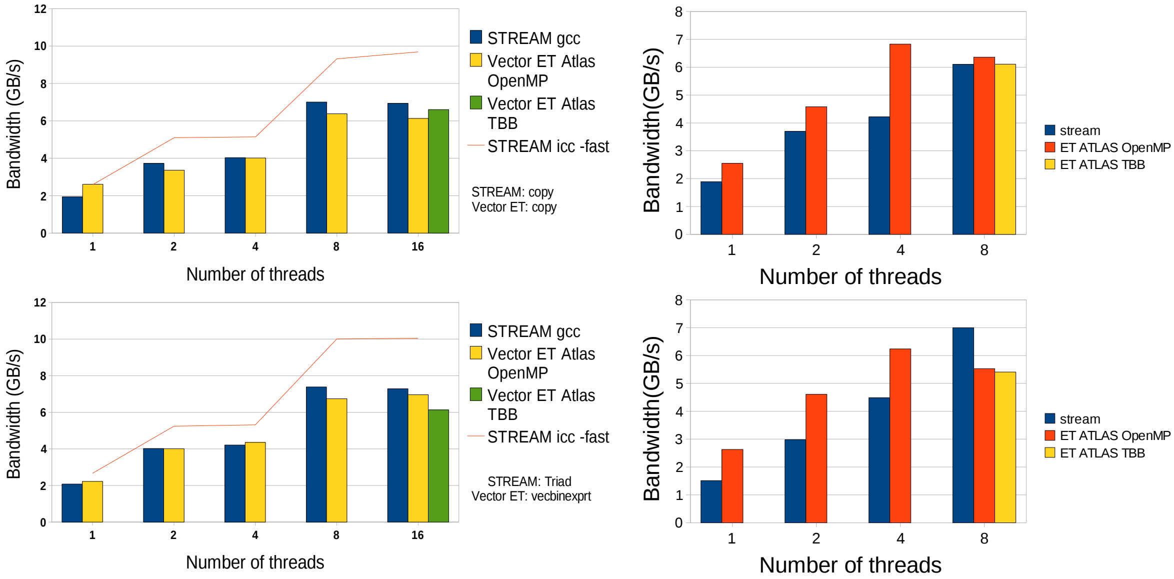


FIGURE 4.6 – Sustained memory bandwidth for out-of-cache vector expressions using the OpenMP C code (Stream), Parallel Blocked ET `Vec` class using OpenMP and TBB, depending on the number of threads used. Top results correspond to copy operation (one read), bottom to a binary vector expression (two reads). Results on the left are for a 16 core Xeon architecture, on the right for a 8 core Opteron architecture.

formulation and the parallelism, since it is no more explicitly described but directly embedded in the expression template tasks.

The apparent memory bandwidth obtained does not correspond to the peak memory bandwidth of the machine, and changing for instance the compiler (considering icc instead of gcc on the Intel machine) would lead to some further performance improvement. Depending on how the assembly instructions are scheduled, cycles taken in the decoding phase of the program execution and usage of the functional units will differ (removing possible stalls for instance). Moreover, usage of non-temporal moves will affect also the number of accesses to caches and can improve performance.

4.1.6 Short recall

This section proposed a short and simple way of improving performance for vector operations, using a high level description. Obtained out-of-cache performance are similar or better than OpenMP C code, and outperforms most C++ libraries when out of cache. This method allows to introduce parallelism inside an initially sequential task graph, embedding the parallelism inside each task (more or less) independently.

The next section will present the other way to parallelize a sequential task graph. According to the dependence between these tasks, they will be spread and schedule across a parallel machine, with the computation embedded in each task remaining sequential.

4.2 Parallel scheduling of sequential tasks

The second part of this chapter proposes a new approach for parallel tasks scheduling on heterogeneous platform.

Modern high performance computing systems contain multiple processing elements. These processing elements may be of similar nature or may have diversity in their processing capabilities (e.g. multicores and accelerators, cores with different frequencies, ...). The inherent parallelism of an application may be exploited by making an effective use of these architectures. The parallel execution of several tasks/modules of an application results in improved performance. In this regard, an effective scheduling of the tasks on the existing processing elements may significantly impact the performance.

Finding optimal scheduling of tasks executing on parallel systems is NP-complete. In general, the scheduling problem is concerned with finding the allocation of tasks to the processing elements and the sequence in which these tasks should be executed. The heterogeneity further adds to the complexity as the processing elements possess diverse processing power. For scheduling, an application is divided into several tasks represented by nodes in a Directed Acyclic Graph (DAG). The edges in the DAG represent dependencies between tasks. The edges are labelled with the communication overhead incurring between tasks of the application. An edge is also a manifestation of the precedence constraint so that a task can not execute before completion of its predecessors. The scheduling then assigns tasks to the processors and also orders the execution of the tasks so that the overall execution time of the tasks is minimum.

Our new scheduling approach, called Minimum Strided Finish Time (MSFT), is presented in this section, organized as follows : Subsection 4.2.1 presents the existing scheduling heuristics, and outline the difference between them and MSFT. Subsection 4.2.2 describes the main context within which the entire approach works. The MSFT approach is formally described and elaborated in Subsection 4.2.3. The experimental results are presented in Subsection 4.2.4. As said before, Subsection 4.2.5 provides an overall conclusion regarding contributions of each works and their interactions with the previous chapters.

4.2.1 Existing scheduling heuristics

The scheduling of modules of an application is a key factor in achieving better performance. In the past, various algorithms and heuristics have been proposed which may be used to facilitate both, the static and the dynamic scheduling. The static scheduling approach generates schedule that is known at compile time. In contrast, dynamic scheduling produces schedules during execution of the application by adapting the schedule with respect to the runtime behavior of the application.

The static scheduling approaches may be further categorized into list/priority-based, cluster based, random-search based and task duplication based heuristics.

List-Based Scheduling Heuristics

The list-based heuristics [97, 91, 61, 71, 121, 43, 3, 64, 94, 65] generate schedule by first assigning priorities to the tasks, and then selecting the best processor that optimally fits the criteria defined by the strategy.

The Heterogeneous Earliest Finish Time (HEFT) Algorithm

The HEFT algorithm [97] works in two phases, with the first phase assigning ranks to tasks and and the secod phase assigning processors to tasks based on *Eearliest Finish Time.*

The ranks are recursively computed for all the nodes of the graph. The rank is dependent on the average of execution costs of the node, together with the communication costs and rank of its successors. The tasks are sorted with respect to the ranks and scheduled subsequently.

The *Eearliest Start Time* represents the time at which the execution of a node on a specific processor may start. It takes into account the processor availability time and the finish time of the predecessors of the node being considered. The *Earliest Finish Time* represents the sum of execution costs of a node on the processor and the *Earliest Start Time*. The HEFT algorithm schedules the node with the processor that produces the minimum *Earliest Finish Time* value.

For $|V|$ tasks, $|E|$ edges and n processors, the complexity of the HEFT algorithm is $O(|E| * n)$ or $O(|V|^2 * n)$.

The Modified Critical Path (MCP) and Mobility Directed (MD) Scheduling Heuristics

The MCP and MD heuristics [121] make use of *as-soon-as-possible ASAP)* and *as-late-as-possible (ALAP)* attributes of the input graph to schedule tasks for a bounded number of processors. The $ALAP$ values are calculated for each node and all the descendants of each node. For each node, a list of $ALAP$ values is created, and subsequently sorted. The nodes are then arranged in order with respect to the values in the lists. This is followed by scheduling of sorted nodes, which assigns a processor to a node that results in earliest execution. The algorithm works with the complexity of $O(|V|^2 log|V|)$.

The MD heuristic minimizes the number of processors required for scheduling. It makes use of relative mobilities calculated as the ratio of the difference of $ALAP$ and $ASAP$ values to the execution time of the node. Once the relative mobilities are computed for each node of the graph, the group of nodes are found and arranged with respect to relative mobility. For each node of the selected group, the scheduling then proceeds by checking for each processor against the pre-defined constraints. The relative mobilities are updated, and the corresponding node is removed from the group. This process iterates for all the groups until there are no more nodes to be scheduled. The MD algorithm has the complexity of $O(|V|^3)$.

After scheduling, the virtual processor set is mapped to physical processor set while minimizing the total communication cost.

Clustering Algorithms

The clustering heuristics [119, 58, 25, 26, 49] map the collection of tasks to processors. A cluster is formed by taking into account the edge weights and linearity of tasks. These clusters are then refined by merging the clusters in multiple steps. Clustering algorithms although make use of a broader view of the graph, but these algorithms target homogeneous architectures and are considered to be more costly than the list scheduling algorithms.

The Dominant Sequence Clustering (DSC) Algorithm

The DSC algorithm [119] incorporates computation of *tlevel* and *blevel* values for each node. For a node v, the *tlevel* is the length of the longest path from entry node to the node v. Similarly, the *blevel* is the length of the longest path from the node v to the exit node.

All nodes are considered to be the part of an un-examined group that is updated whenever a node becomes part of some cluster. Initially, every node belongs to a unit cluster. The un-examined nodes are processed in order of the priorities assigned to them. A node may then be merged with the cluster of any of its predecessors so that the *tlevel* value of that node decreases optimally. If there is an increase in the *tlevel* value, the node remains to be the part of the unit cluster. Subsequently, the priorities are updated and the node is removed from the un-examined group.

The complexity of the DSC algorithm is $O((|V| + |E|)log|V|)$, where $|V|$ is the number of tasks and $|E|$ is the number of edges in the graph.

Triplet Algorithm

The Triplet algorithm [26] is also a clustering algorithm that facilitates assignment of clusters of tasks to clusters of workstations. It uses the concept of triplets, where a triplet represents three tasks. These triplets are sorted with respect to the degree and the communication cost. Subsequently, the clusters are merged and mapped to the clusters of workstations.

For each cluster of tasks, the tasks are arranged in order with respect to the code size and the communication cost. The workstations having very close communication bandwidth and processing power are placed in a single cluster of workstations. For each cluster of workstations, the clusters are arranged with respect to the communication bandwidth and the processing power. While mapping, a workstation from another cluster of workstations may only be assigned when the number of operations assigned to the current cluster of workstations exceeds the load calculated statically. The workstation that results in the best completion time is then assigned to the cluster being considered.

The algorithm has the complexity of $O(|V| * D * log(D) + n * |V|)$, where $|V|$ is the number of tasks, n is the number of processors and D is the product of the maximum in-degree and out-degree of the graph.

Task Duplication Algorithms

The task duplication heuristics [33, 37, 2, 81, 19, 63, 23] are based on the intuition that effective schedules may be produced by reducing the inter-processor communication and utilizing idle times of the schedules, which can be achieved through duplication of the appropriate nodes on other processors. Similarly to clustering algorithms, the task duplication algorithms have higher complexity with no guarantee of producing optimal schedule.

Levelized Duplication Based Scheduling (LDBS)

The LDBS algorithm [37] schedules tasks by duplicating them at various levels of the graph. The levels of a graph are generated by level sorting. LDBS algorithm calculates rank of a node based on the length of critical path that represents the path with largest sum of execution and communication costs.

The LDBS algorithm has two varaints : LDBS1 and LDBS2. In first variant LDBS1, the Earliest Start Time and Earliest Finish Time are computed. For each processor, the start time of a task node is minimized by duplicating its immediate predecessor tasks. After duplication, the task is assinged to the processor that results in minimum start time of execution of the task being considered. Using this variant, all the nodes at a level have the same likelihood of being processed for scheduling regardless of their existence/non-existence on the critical path. The second variant LDBS2, in contrast, first arranges tasks with respect to the ranks. Each task is then considered for scheduling with respect to the rank assinged to it. Consequently, the tasks on the critical paths are scheduled before other tasks.

The LDBS1 variant of the algorithm has the complexity of $O(|V|^3 * |E| * n^3)$, whereas the LDBS2 version has the complexity of $O(|V|^3 * |E| * n^2)$, for $|V|$ tasks with e edges in the DAG, and n processors.

4.2.2 Context of the MSFT Scheduling Approach

The scheduling approach developped in the following subsections, called Minimum Strided Finish Time (MSFT), falls in the category of list scheduling algorithms. However, it takes into account a broader view of the task graph while working with low complexity. Its cost function is based on considering the children of the nodes lying within a fixed stride. The decision of this augmented cost function makes it possible to produce reduced schedule length as compared to other list scheduling strategies.

Multiple processing elements may be fully exploited if the parallel tasks of an application are scheduled in an effective way. The input to a scheduling algorithm is a Directed Acyclic Graph (DAG) G = (V, E), where V is the set of nodes representing tasks, and E is the set of edges representing dependencies among the tasks. These edges are labelled with the communication cost occuring among the tasks.

For scheduling tasks in the graph, we add two nodes into the graph : *Start* and *Exit* nodes. The *Start* node has no predecessors, and the *Exit* node has no successors.

For the scheduling algorithm suggested in this article, we make the following assumptions :

1. There are no communication costs between two tasks executing on the same processor.
2. The *Start* node and the *Exit* node have no communication and execution costs associated with them.
3. There is a fixed number of heterogeneous processors which are connected in such a way that communication may take place through communication channels among any of the processors, i.e., they are connected through mesh topology.
4. A task may only start execution when all its predecessor tasks have completed their execution.
5. A processor may execute the task and communicate with other processors at the same time.
6. A task is executed only once, by one processor (no task duplication).

The MSFT algorithm works for m number of tasks, and n number of processors. Let $ET(T_i, P_j)$ represent the execution time or latency of a task T_i on processor P_j. Similarly, let $CC(T_k, T_i, P_x, P_j)$ represent the communication cost incurring due to data transfer from a task T_k that has been assigned a processor P_x to another task T_i, if the latter task is executed on processor P_j. If P_x and P_j are the same processors, the communication cost is assumed to be zero.

Let $ImPred(T_i)$ represent the immediate predecessors and $ImSucc(T_i)$ represent the immediate successors of a node T_i. Similarly, the notation $RSucc(T_i)$ is used to represent the lists of successors of a node existing within a stride (or distance) R from the current node. While scheduling, the possible available time of a processor P_j is represented by $Avail(Pj)$. It is updated after each assignment of a task to the processor. The algorithm proceeds in two phases : in the first phase, the prioritization of tasks taskes place, and the second phase assigns processors to the tasks as elaborated in this subsection.

Prioritization of Tasks

The MSFT algorithm first assigns priorities to the tasks. Consequently, the tasks are arranged in order so that the task with the highest priority is processed first for assignment. The priority of the tasks is computed recursively, and is based on the Earliest Execution (EE) times as given below.

Computation of the Earliest Execution (EE) Times

The Earliest Execution (EE) time corresponds to the earliest time when a task can be scheduled, taking into account its dependencies (the tasks it depends on) and considering the best mapping to processors. Tasks are ordered according to their EE values in topological order.

Computation of Priorities

The computation of the Earliest Execution times is followed by assignment of priorities to the tasks. For each task, the priority is based on the minimum values of the Earliest Execution time for the processors as calculated in the previous step. The tasks will be processed with respect to the priorities assigned to them. According to the definition of priorities, it is straightforward to show that the resulting schedule is valid w.r.t. dependencies.

Processor Assignment

The processor assignment makes use of the Strided Finish Time (SFT) and the Actual Finish Time (AFT) as described here.

Computation of the Strided Finish Time

While scheduling the tasks, the Strided Finish Time (SFT) is calculated, which represents the finish time of a task taking into account the execution cost of its successors. Depending upon the processor availability, a task may start execution after completion of its predecessor tasks and the transfer of data. Let $ST(T_i, T_k, P_j)$ represent the start time of a task T_i with respect to the predecessor T_k on processor P_j. Subsequently, the Earliest Start Time $EST(T_i, P_j)$ of the task T_i on processor P_j is computed followed by computation of the Strided Finish Time $SFT(T_i, P_j)$ of the task T_i on processor P_j.

Intuitively, if a task is assigned to a processor, say P_j, the SFT value with stride equal to one would represent the finish time of this task and its immediate successor, using the same processor P_j. With multiple immediate successors, the largest values are selected corresponding to each processor. The processor producing the minimum SFT value is then assigned to the task. The stride may be increased if we need to consider successors up to a specific level.

Actual Finish Time

The SFT value for a task need not be the same as the Actual Finish Time (AFT), which is the finish time of the task after the processor assignment for the task has taken place.

AFT is calculated for each task being scheduled with initial value of entry node being zero. It is computed by taking into account the dependence constraints and architecture constraints. The total schedule length is the largest value of AFT for any node in the task graph.

4.2.3 Minimum Strided Finish Time (MSFT) Algorithm

The first phase of the MSFT scheduling strategy assigns priorities to the tasks by taking into account the immediate predecessors of the tasks as given in Algorithm 2. The nodes of the graph are processed in topological order. The Earliest Execution (EE) time is iteratively computed. Step 4 computes the EE for a task T_i on a processor P_j as the maximum value among the minimum sum values obtained by

Algorithm 2 MSFT algorithm phase 1 : prioritization of tasks

```
//Let $m$ be number of nodes/tasks and $n$ be the number of processors
Initialize $EE \leftarrow 0$.
for $i = 1$ to $m$ in topological order do
    Compute the earliest execution ($EE$) time of a task $T_i$ according to the earliest execution times of its predecessors and depending on the processor executing $T_i$ :
        $$EE(T_i, P_j) = \max_{T_k \in pred(T_i)} \min_{v=1..n} \left( EE(T_k, P_v) + CC(T_k, T_i, P_v, P_j) \right) + ET(T_i, P_j). \quad (4.3)$$
    Compute task priorities by using the $EE$ values found in previous step.
        $$Priority(T_i) = \frac{1}{\min_{v=1..n} EE(T_i, P_v)}. \quad (4.4)$$
    In case of equal priority, the topological ordering of the input nodes is used.
end for
```

adding the EE of immediate predecessors of T_i, the execution latency of the task and the communication time involved. The EE resulting from this computation provides a schedule time for each task compatible with dependence constraints. The task priority is computed by finding the minimum EE value for the task corresponding to any of the processors, as described at step 5 of the algorithm.

To take into account the fact that one processor cannot execute more than one task at a time, the tasks are ordered w.r.t. the priorities assigned to them. It also ensures that a task with a higher priority is processed earlier for processor assignment as described below.

Algorithm 3 describes the processor assignment phase of the MSFT scheduling approach with stride equal to 1. The steps 1-3 initialize the actual finish time (AFT), describing the completion time of a task, the $Avail$ array, describing for each processor the time where it is available using insertion based strategy, and the mapping function M, mapping tasks to processors. The start time for a node w.r.t. its predecessors is computed at steps 7-9. The Strided Finish Time is then computed at step 11 which takes into account the Earliest Start Time (EST), execution time (ET) of the current node and the execution time of the immediate successor nodes. The processor producing the minimum Strided Finish Time (SFT) is assigned to the task at step 12. In case, two or more processors return the same SFT value, the assignment would be made to the fastest processor (having smallest execution latency for the task). The actual finish time and the $Avail$ array are subsequently updated.

The criteria of selecting the successor nodes at stride one may be extended to larger strides by replacing Equation 4.7 (computing the SFT value) with the following equations.

$$SFT_0(T_i, P_j) = \max_{T_{l1...R} \in RSucc(T_i)} \Big(ET\left(T_{l1}, P_j\right) + ET\left(T_{l2}, P_j\right) + \ldots + ET\left(T_{lR}, P_j\right) \Big), \quad (4.9)$$

$$SFT(T_i, P_j) = EST\left(T_i, P_j\right) + ET\left(T_i, P_j\right) + SFT_0\left(T_i, P_j\right). \quad (4.10)$$

Algorithm 3 MSFT algorithm phase 2 : processor assignment with stride 1

```
Initialize $Avail(P_j) \leftarrow 0$, for each processor.
Initialize $AFT(T_k) \leftarrow 0$, for each task,
Initialize $M(T_k) \leftarrow 0$, for each task,
// Let $m$ be the number of nodes
for $i = 1$ to $m$ do
    $T_i \leftarrow$ highest priority task in the list.
    for each task $T_k \in ImPred(T_i)$ do
        $ST(T_i, T_k, P_j) = \max\Big(Avail(P_j), AFT(T_k) + CC(T_k, T_i, M(T_k), P_j)\Big).$    (4.5)
    end for
    Compute the Strided Finish Time (SFT) for each processor $P_j$.
        $EST(T_i, P_j) = \max_{T_k \in ImPred(T_i)} ST(T_i, T_k, P_j)$    (4.6)
        $SFT(T_i, P_j) = \max_{T_k \in ImSucc(T_i)} \Big(EST(T_i, P_j) + ET(T_i, P_j) + ET(T_k, P_j)\Big)$    (4.7)
    Map $T_i$ to the processor producing the smallest value of SFT :
        $M(T_i) \leftarrow \text{argmin}_{P_j} SFT(T_i, P_j)$
    Update the Actual Finish Time($AFT$) for task $T_i$ when executed on processor $M(T_i)$.
        $AFT(T_i) = ST(T_i, M(T_i)) + ET(T_i, M(T_i)),$    (4.8)
    Update the availability time $Avail(M(T_i))$ of processor $M(T_i)$.
    Remove the task $T_i$ from the list.
end for
```

Using equations 4.9 and 4.10, the execution times $T_{l1}, T_{l2}, \ldots, T_{lR}$ of the critical successors of the node T_i (existing at each level) up to level R are taken into account for computing SFT.

Complexity of the MSFT algorithm

The complexity of the MSFT algorithm is calculated for both the phases using the task graph containing $|V|$ nodes, $|E|$ edges and having n number of processors.

For the first phase, the Earliest Execution time is computed in $O(|V| * degree_{in} * n^2)$. The priority computation has a complexity $O(n * |V|)$ respectively. So, the prioritization phase has the overall complexity of $O(|V| * degree_{in} * n^2)$, or $O(2 * |E| * n^2)$. The second phase with stride one has the complexity of $O(|V| * n * (degree_{in} + degree_{out}))$, or $O(|E| * n)$.

For stride equal to R, the complexity becomes $O\left(|V| * n * \left(degree_{in} + \sum_{i=1}^{R} C_{degree_{out}}(i)\right)\right)$, where $C_{degree_{out}}(i)$ represents the out-degree of the most critical node existing at

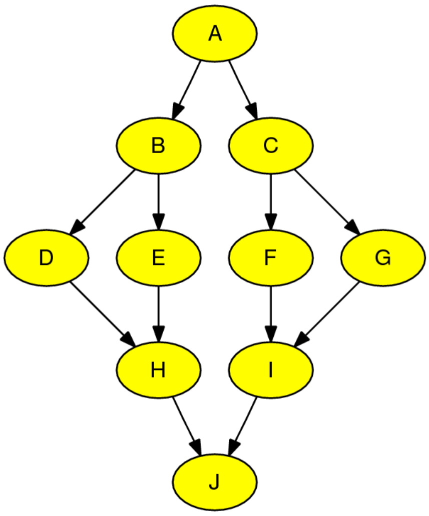


FIGURE 4.7 – Fork-join graph (degree=2, width=2 and depth=1)

stride i.

4.2.4 Experimental Results

Different graph topologies including fork-join, random, gaussian-elimination, out-tree and in-tree graphs have been used for experimentation. Only fork-join and random graphs are displayed in this chapter. We use the parameters depth and width to refer respectively to the number of subgraphs of a particular type vertically and horizontally. The degree parameter represents the degree of the subgraphs used in the task graph. Figure 4.2.4 shows the fork-join graph with degree=2, width=2 and depth=1.

The experimentation has been performed with a large number of parameters. The execution latencies of the tasks have been assigned random values for all the processors. The communication costs are computed by using a set of CCR (communication to computation cost ratio) values. Two common parameters, CCR and the number of processors, are used for all topologies of the graphs as given below, whereas other parameters for each graph are given in their respective subsections.

CCR={0.5,1,2,3,4,5,6,7,8,9,10}
Number of processors = {2,3,4,5,6,7,8,9,10,12,14,16}

A metric SLR (Schedule Length Ratio) is used to measure the effectiveness of the schedule. It represents the ratio of the schedule length to the length of the critical path found with the minimum execution costs, i.e.,

$$\text{SLR} = \frac{Schedule\ Length}{Length\ of\ Critical\ Path\ with\ minimum\ execution\ latencies}.$$

The length of the critical path is the lower bound on the schedule length of a task graph. Since there are a large number of graphs for each topology used for experimentation, the average SLR values are computed and presented in the results.

Fork-Join Graphs

The fork-join graph generator takes the depth, degree and width parameters to generate nodes of the graph. For a given depth m, degree n and width w, the number of nodes produced by the graph generator is equal to $(m*(n+1)+1)*w+2$. The configuration parameters for the fork-join graphs are given below.

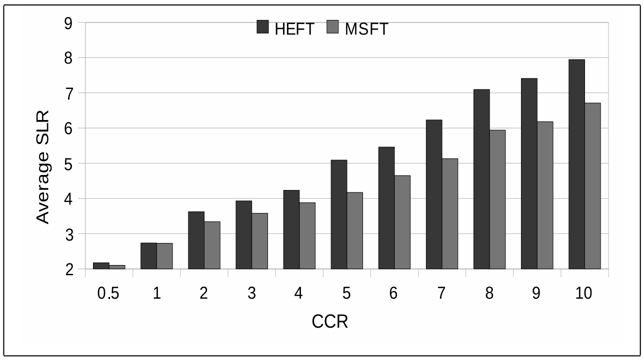


Figure 4.8 – Results of the Fork-Join graphs for the given CCR values

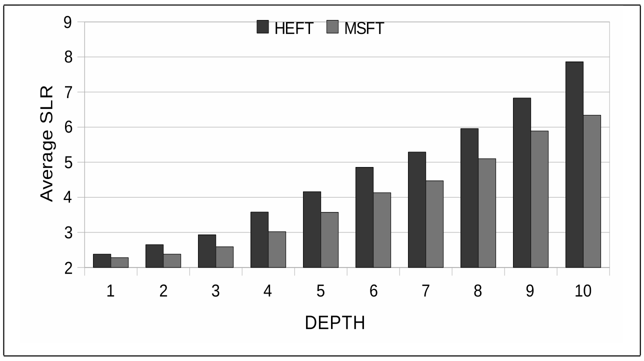


Figure 4.9 – Results of the Fork-Join graphs for the given depth values

Depth={1,2,3,4,5,6,7,8,9,10}
Width={2,3}
Degree={2,3,4}

The scheduling results of fork-join graphs are shown in Figures 4.8, 4.9 and 4.10 for various values of CCR, depth and the number of processors respectively. The MSFT scheduling strategy performs better than the HEFT scheduling strategy for all the cases with an overall improvement of 12.29%.

For some cases of small CCR values, the HEFT performance is very close to MSFT, however the MSFT schedule improves gradually with an increase in the CCR value, as shown in Figure 4.8.

With the growing number of nodes, the MSFT strategy tends to produce better schedule than HEFT as shown in Figure 4.9. This behavior is similar for various number of processors. As shown in Figure 4.10, the SLR value although increases with an increase in the number of processors, the MSFT schedule is always better than the schedule produced by HEFT.

Random Graphs

The random graphs are generated using the approach given in [5]. An edge between two nodes exists based on the probability in such a way so that the precedence constraints are not violated. Therefore, the edge from a node i to node j exists if $i < j$. Given the probability p, there is a unique edge from a node i to the node $\lceil (i + 1/p) \mod m \rceil$, where m is the number of nodes.

The random graph generator takes the number of nodes and the probability as additional parameters. The data set used for experimentation is as follows.

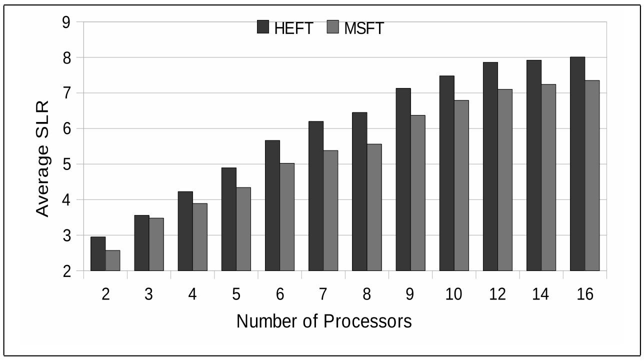


FIGURE 4.10 – Results of the Fork-Join graphs for the given number of processors

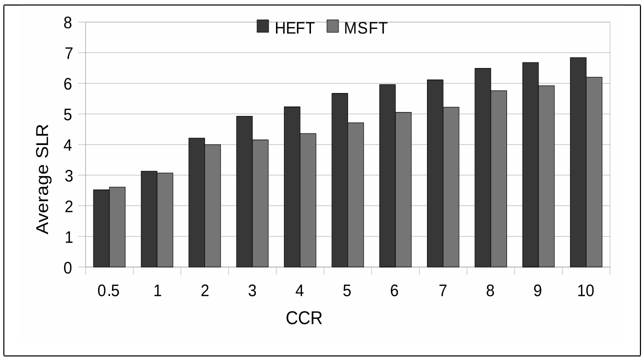


FIGURE 4.11 – Results of the Random graphs for the given CCR values

Number of nodes={10,20,30,40,50,60,70,80,90,100}
Probability={0.2,0.4,0.5,0.6,0.8,1.0}

The impact of CCR values on the average SLR varies as shown in Figure 4.11. For very small and large values of CCR, the SLR values are close for both the scheduling strategies. However, for medium size values of CCR, the MSFT scheduling strategy performs better than the HEFT scheduling strategy.

With the increasing number of nodes, the MSFT scheduling performs better than HEFT as shown in Figure 4.12. Similarly, for a large number of processors, MSFT generates schedules with improved SLR compared to those generated by HEFT as shown in Figure 4.13.

For the random graphs, the MSFT strategy produces an average improvement

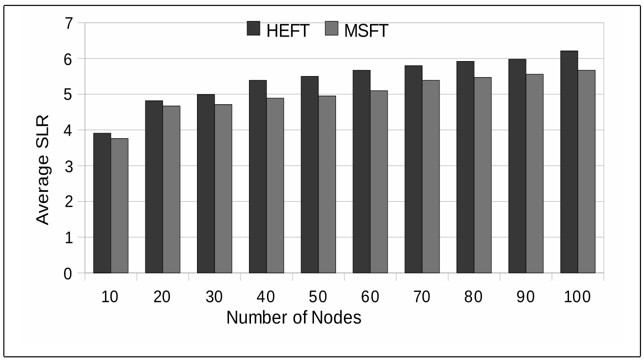


FIGURE 4.12 – Results of the Random graphs for the given number of nodes

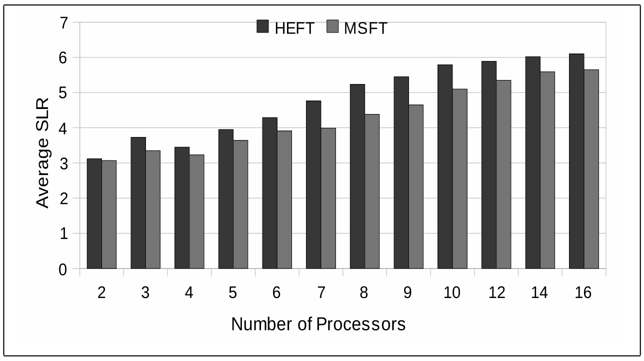


FIGURE 4.13 – Results of the Random graphs for the given number of processors

of 9.78% over the HEFT strategy.

4.2.5 General Conclusion

This chapter presented two works for automatic vertical and horizontal parallelization, widening the contribution range of the methods developped in Chapters 2 and 3.

In the first part of this chapter, C++ Expression Templates mechanism is used to implement a parallel version of vector operations, using Intel TBB and OpenMP. This vertical parallelization through expression templates provides a short and simple way of improving performance for vector operations in the C++ language, outperforming most C++ librairies when out of cache. Moreover, the study on Expression Templates can be coupled with the CPC framework, the original vector operation serving as a base for it, and the result of the framework providing better basic blocks when implemnting the expression templates.

The second presented work proposes a novel approach of task-graph scheduling called Minimum Strided Finish Time (MSFT), which generates schedules for heterogeneous systems. The MSFT algorithm takes into account the current node being scheduled together with the nodes existing within a specific distance in the graph (the stride), first prioritizing the nodes by computing their earliest execution time, then scheduling each tasks according to its priority. A large number of experiments have been performed for various graphs including the fork-join, gaussian-elimination, in-tree, out-tree and random graphs. The complete work is currently revised for a future re-submission, with results showing a significant reduction in the schedule length compared to the well-known HEFT scheduling algorithm. Increasing the stride used by our method is an efficient way to improve the schedule, at the expense of an increased complexity. The impact is however limited in practice because MSFT with small stride value already produces near-optimal results on all graphs tested.

Furthermore, combining this scheduling method with the CPC framework, and adding some way to describe the dependence graph between codelets, one will be able to find the hot-spot loop in a program, decompose it in several codelets, optimizing them, then schedule the codelets on a multicore machine to obtain the best performance. More specifically when dealing with stencil codes, using the code generation model describe in Chapter 3, one can get efficient code for part of the stencil for each node in an heterogeneous machine using CPU and GPU, then find the best schedule using MSFT algorithm.

Chapitre 5

Conclusion

With the permanent development of processor architectures, the methods presented in this dissertation will evolve to take into consideration the emerging and future concerns of the HPC community. In this last part, we will recall the contributions of this dissertation, before discussing their possible changes to answer the modern issues in computer science, as the power consumption or manycore processors.

5.1 Contributions

This dissertation presents some solutions to answer the challenges of generating efficient irregular codes and efficient parallel codes, including parallel codes for heterogeneous systems. The two main contributions are source-to-source transformations, allowing to take any source code produced by a programmer with potentially no knowledge of the target architecture, and to generate a new code exhibiting good performances :

- The CPC framework allows to optimize a first implementation of large programs (like simulation codes) by applying transformations only on a subset of the program.
- Multi-Padding modifies the data layout of stencil codes to decrease the number of alignment conflicts in those codes.

The CPC framework proposes to extract codelets from nested loops of hot-spot region in a real-life program, in order to optimize independently each codelets without having to run the all program for each optimization attempt. Loop transformations modify the codelets, and each version is evaluated. The description of the optimization space is realized through user defined pragmas. The hot-spot function is then recomposed with the best codelets, and the overall speed-up is predicted from the codelets performance. The codelets are evaluated in a fixed memory configuration, and prefetching or cache cleaning blocks are added to ensure that data in the overall computation are in the same memory configuration than during the tests. The methods have been applied successfully on three real programs used in theoretical physics, chemistry and bioinformatics.

The second main contribution proposes several methods based on padding to address alignment issues on linear algebra codes using arrays, and more specifically stencil codes, which are symptomatic of alignment issues. From the original access pattern in an array, the several methods generate a linear system to resolve in order to align a maximum of data. Each method correspond to a different trade-off level between the complexity of the linear system to resolve and the number of data naturally aligned when generating the new code. The simpler the method, the fewer the number of padding values, and the lesser the number of data well aligned. The simplest method with only one padding value correspond to the usual padding methodology, while the most complex system is formed when having a different padding value for each line of the studied array. We show that alignment issue, whether for efficient vectorization or limiting bank conflicts, on CPUs and GPUs can be addressed the same way, and an algorithm is provided to generate efficient stencil codes for CPUs and GPUs using our alignment methods.

The last chapter proposes two distinct methods to address the two parallelization techniques : vertical and horizontal parallelization. Vertical parallelization consists in separating an iteration domain in several independent slices, which will be executed simultaneously in parallel on several cores. We use C++ expression templates to represent vector and array computation, delaying their evaluation to where they are actually need. The actual implementation of said computations is directly parallelized to run on the several cores available. We end up with a sequential dependence graph of parallelized tasks. Horizontal parallelization decomposes a program on several tasks linked through a dependence graph, each tasks in the same level of the graph may run in parallel. The described method compute the earliest finish time of a task, choosing on which available unit it must run, on an homogeneous or heterogeneous system. To take its decision, the algorithm takes into account the following tasks in a fixed number of graph levels, pre-evaluating each choice. This stride allow a better evaluation of the overall earliest finish time, but the number of evaluations increases exponentially for each level added in the prevision, and once again, the overall speed-up will depend of the computation time/power allocated to the optimization process.

5.2 Future works

Most of the contributions propose methods to automatically generate efficient codes from an initial implementation, but very few steps are nowadays realized in an automatic way.

In the CPC framework, only the generation of multiple codelets version from an original extracted codelets is perform automatically, and even in this step the user intervenes to define the transformation space. Numerous tools already exist to perform all parts of the framework : profiling tools to find the hot-spot function of a program, code parser and code extractor to find the problematic loop nest and extract several codelets from this loop nest, loop transformations performed with X-language. To fully automatize the optimization process, we can couple the X-language framework with some tree-pruning algorithm, beginning with a large number of possible value for each transformation, and cutting the intervals when more speed-up is possible. A code generator can be easily added to generate an evaluation routine calling all the transformed codelets to evaluate. Writing all the results in an output file, it is simple for a sorting routine to find in it the best version for each codelets. A last pass from the code generator will produce the new function calling the best kernels.

A first stencil code generator was realized, producing first only stencil code for double precision Jacobi. Since, the generator was modified to generate codes for any stencil pattern, for any types of data. The padding values found by solving the linear system are specified by the user. With the existing solver library, like CPLEX or lpsolve, it is possible to create the three modules in the same language to realize all steps in a same program. A library could gather all the proposed methods implementation. The first module will identify the stencil pattern, either reading explicitly the pattern in an input, or from a pattern recognition of an initial implementation, then select which method to apply on the pattern to produce the linear system. The second module will simply resolve the system, and send all padding and shuffle values to the third module which will generate the new better aligned stencil code.

Some combinations between works presented independently in this dissertation seem to be promising. When evaluating an expression template, an actual implementation is called to perform the computation. Instead of dealing with a basic implementation of the vector or array operation, one may take the time to apply first the CPC framework in the basic implementation to generate a better version, replacing the basic implementation when evaluated. Also, when applying the CPC framework on some code, on can consider extracted codelets as independent tasks, which can be executed in parallel, and on different architectures. The MSFT algorithm can then be applied on the codelets, with the only additional cost of dependence generator when extracting the codelets, and having to evaluate all codelets not only on the original CPU but on all available devices.

Future horizons

Apart from future works directly related to automatize or implement part of the methods presented in this dissertation, current issues in the high performance computing arena offers new horizons for optimizations. Until the last decade, and except for embedded systems, optimization in HPC was synonym of speed-up, reducing the time spent to run a program. Nowadays with the power consumption being a real issue, scientists are willing to take more time to execute a code, if it enables energy savings [93, 8]. This consideration will again increase the complexity of producing an efficient program, creating yet another paradigm shift, in programming models this time. We shift from one goal, only time speed-up to consider, to several issues to address in the same implementation, reducing the time and the power consumption.

With the constant increasing complexity of computer architectures, it will be more and more difficult to create software and methods efficient for all codes, for all targets. Before multi-core chips, compilers were already overtaken, like for SIMD instructions which could be addressed only through very explicit directives. Nowadays, to obtain performance on a machine, in term of time or energy, the programmer will have to know and understand more and more mechanisms. Some scientists prefer to write directly in assembly code or with intrinsics functions whether than understand all the options to trigger for the compiler to generate an efficient code. Optimization softwares to help generate efficient codes will be more and more required, whether they are highly specialized on a machine, or addressing the same detail on all architectures.

The Microkernel-Description-Language based Performance Evaluation Framework (**MDL-PEF** [45]) aims the latest consideration. It extracts the data flow structure of an assembly code, and translate it in its own representation. A predictor uses pattern matching to compare the intermediate representation to a MDL-Microkernel database for predicting performance. Each codelet is linked to a canonical representation, and a database is available for each targeted architecture. Hence, from the assembly code for a specific architecture, one will be able to find microkernels doing the same computations on other targets. MDL-PEF provides a tool to initialize a pattern matching database for the target architecture.

The coming of manycore architectures accentuates existing issues, like concurrent accesses to the main memory, contention on the controller bus, tuning and spread of the computation on multiple cores and architectures. With chips embedding more than forty cores, and each cores (or couple of cores) accessing its own private memory, data distribution and message passing will become even more important. These manycores chips, like Intel Single Cloud Computing processor or

IBM's Cyclops 64, can be seen as hardware accelerators. The numerous embedded cores have simplified architecture designs, they are controlled by a front-end machine and they have each their own private local storage. Processors becoming some sort of hardware accelerators, connected themselves to other hardware accelerators (like GPUs), the need for models and tools to efficiently program such machines in an automatic and heterogeneous way is more and more paramount.

Annexe A

N-D Jacobi Stencil using vector larger than two elements

In this annex are presented the equations for aligning the required elements in a cell update with a N-D Jacobi stencil pattern. Heavy detailed equations are presented in this section, to better understand the principles of ***MDMP*** on a specific example. These equations are easy to simplify. The elements highlighted in red in the following systems of equations are the common part between the two equations, cancelling themselves out.

A.1 Detection of necessary padding, with vector of any size

Let us consider that in this section, for all dimensions, there is an even number of hyperplanes. Expanded the work done in section 3.3.3 for any number of dimension, the following suppositions are made for the proofs by induction :

- For the k firsts dimensions, the even hyperplanes composing it have a size $S^e_{k-1} \equiv_v 1$, v being the vector size.
- For the k firsts dimensions, the odd hyperplanes composing it have a size $S^o_{k-1} \equiv_v v-1$.
- For each dimension k, a value $offset_k$ exists such as $S^e_k + offset_k = S^o_k - offset_k$. The resulting average dimension size is named S_k.

To simplify already complex systems of equation, a new function Dim_k is defined to return a dimension size, regarding the sizes if its odd and even hyperplanes, and considering if there is an odd or even number of hyperplanes in the dimension. Fig.A.1 presents the function definition.

$$Dim_k = \begin{cases} \frac{(S^e_k + S^o_k) * \frac{S_{k+1}}{2}}{S_{k+1}} \text{ if } S_{k+1} \equiv_2 0 \\ \\ \frac{S^e_k * \left(\lfloor \frac{S_{k+1}}{2} \rfloor + 1 \right) + S^o_k * \lfloor \frac{S_{k+1}}{2} \rfloor}{S_{k+1}} \text{ if } S_{k+1} \equiv_2 1 \end{cases}$$

FIGURE A.1 – Function returning the average size of a dimension, depending on the sizes of its odd and even hyperplanes

Since the average dimension size S_k will be used in the following complex system of equations, the proof that the *offset* value is still transparent will be shown first.

In equations A.1 and A.2, *offset* is included in the affected parts of an indexed cell, respectively for a even hyperplane and an odd hyperplane.

$$1+(i_{k+1}+1)*(S_k^e{\color{blue}+offset_k}+S_k^o{\color{blue}-offset_k}) \tag{A.1}$$

The added *offset* values, highlighted in blue, cancel each other very quickly.

$$(i_{k+1}+1)*(S_k^e{\color{blue}+offset_k})+i_{k+1}*(S_k^o{\color{blue}-offset_k}){\color{blue}-offset_k} \tag{A.2}$$

Considering the suppositions made at the beginning of this section, equation A.3 is the system to solve to compute the value S_k^e for elements $(1,\dots,k,\dots,n)$ and $(0,\dots,k+1,\dots,n)$ to have the same alignment.

$$\left.\begin{aligned}
&\sum_{a=k+2}^{n-1}\left(i_a*\prod_{b=k}^{a-1}Dim_k)*\prod_{b=0}^{k-1}S_b\right)\\
&+i_0+1+\sum_{a=1}^{k}\left(i_a*\prod_{b=0}^{a-1}S_b\right)+(2i_{k+1})*\prod_{b=0}^{k-1}S_b+i_{k+1}*S_k^e+i_{k+1}*S_k^o\equiv_v c\\
&\sum_{a=k+2}^{n-1}\left(i_a*\prod_{b=k}^{a-1}Dim_k*\prod_{b=0}^{k-1}S_b\right)\\
&+i_0+\sum_{a=1}^{k}\left(i_a*\prod_{b=0}^{a-1}S_b\right)+(2i_{k+1}+1)*\prod_{b=0}^{k-1}S_b+(i_{k+1}+1)*S_k^e+i_{k+1}*S_k^o\equiv_v c
\end{aligned}\right\} \tag{A.3}$$

Solving the system A.3, the first part of the proof appears, as $S_k^e\equiv_v 1$ like in our hypothesis (see fig.A.2).

$$(A.3)\iff\left\{\begin{aligned}
&{\color{red}\sum_{a=k+2}^{n-1}\left(i_a*\prod_{b=k}^{a-1}Dim_k*\prod_{b=0}^{k-1}S_b\right)+i_0+\sum_{a=1}^{k}\left(i_a*\prod_{b=0}^{a-1}S_b\right)+(2i_{k+1})*\prod_{b=0}^{k-1}S_b+i_{k+1}*S_k^e+i_{k+1}*S_k^o}\\
&+1\equiv_v c\\
&{\color{red}\sum_{a=k+2}^{n-1}\left(i_a*\prod_{b=k}^{a-1}Dim_k*\prod_{b=0}^{k-1}S_b\right)+i_0+\sum_{a=1}^{k}\left(i_a*\prod_{b=0}^{a-1}S_b\right)+(2i_{k+1})*\prod_{b=0}^{k-1}S_b+i_{k+1}*S_k^e+i_{k+1}*S_k^o}\\
&+\prod_{b=0}^{k}S_b+S_k^e\equiv_v c
\end{aligned}\right.$$

$$(A.3)\iff\prod_{b=0}^{k}S_b+S_k^e\equiv_v 1$$

$$(A.3)\iff S_k^e\equiv_v 1$$

FIGURE A.2 – Size of any even hyperplane for common alignment

To get the second part of the proof, equation A.4 is solved, this time considering elements $(1,\dots,k,\dots,n)$ and $(0,\dots,k-1,\dots,n)$ for a common alignment (see fig.A.3).

$$\left.\begin{aligned}
&\sum_{a=k+2}^{n-1}\left(i_a*\prod_{b=k}^{a-1}Dim_k*\prod_{b=0}^{k-1}S_b\right)\\
&+i_0+1+\sum_{a=1}^{k}\left(i_a*\prod_{b=0}^{a-1}S_b\right)+(2i_{k+1}+2)*\prod_{b=0}^{k-1}S_b+(i_{k+1}+1)*S_k^e+(i_{k+1}+1)*S_k^o\equiv_v c\\
&\sum_{a=k+2}^{n-1}\left(i_a*\prod_{b=k}^{a-1}Dim_k*\prod_{b=0}^{k-1}S_b\right)\\
&+i_0+\sum_{a=1}^{k}\left(i_a*\prod_{b=0}^{a-1}S_b\right)+(2i_{k+1}+1)*\prod_{b=0}^{k-1}S_b+(i_{k+1}+1)*S_k^e+i_{k+1}*S_k^o\equiv_v c
\end{aligned}\right\} \tag{A.4}$$

As expected, $S_k^o \equiv_v v-1$, and the second part of the suppositions is verified. The last part is straightforward, since it was already demonstrated with equations A.1 and A.2. Now that the k^{th} dimension considered have two distinct sizes for its even and odd hyperplanes, these equations apply to it.

$$
(A.4) \iff \left\{\begin{array}{l}
\sum_{a=k+2}^{n-1}\left(i_a * \prod_{b=k}^{a-1} Dim_k(S_b^e, S_b^o) * \prod_{b=0}^{k-1} S_b\right) \\
+i_0 + \sum_{a=1}^{k}\left(i_a * \prod_{b=0}^{a-1} S_b\right) + (2i_{k+1}) * \prod_{b=0}^{k-1} S_b + (i_{k+1}+1) * S_k^e + i_{k+1} * S_k^o \\
+1 + \prod_{b=0}^{k} S_b + S_k^o \equiv_v c \\
\sum_{a=k+2}^{n-1}\left(i_a * \prod_{b=k}^{a-1} Dim_k(S_b^e, S_b^o) * \prod_{b=0}^{k-1} S_b\right) \\
+i_0 + \sum_{a=1}^{k}\left(i_a * \prod_{b=0}^{a-1} S_b\right) + (2i_{k+1}) * \prod_{b=0}^{k-1} S_b + (i_{k+1}+1) * S_k^e + i_{k+1} * S_k^o \equiv_v c
\end{array}\right.
$$

$$
(A.4) \iff 1 + \prod_{b=0}^{k} S_b + S_k^o \equiv_v 0
$$

$$
(A.4) \iff S_k^o \equiv_v v-1
$$

Figure A.3 – Size of any odd hyperplane for common alignment

Our three assumptions have been verified. We proved that if all $(k-1)$ firsts dimensions have two hyperplane sizes congruent to 1 and $v-1$ respectively for even and odd ones, and if an *offset* value can be found for each of its dimension to determine an average hyperplane size, then the k^{th} dimension also have two distinct hyperplane sizes for even and odd ones ($S_k^e \equiv_v 1$ and $S_k^o \equiv_v v-1$, and an *offset* value can be found too.

To find the value of $offset_k$ is also straightforward. The same computation as in section 3.3.3 returns $\forall k$, $offset_k = \frac{v-2}{2}$.

Additional proof for alignment correctness of needed elements in the same dimension Even if the evaluation of the new sizes values S_k^e and S_k^o ensures that the two required elements in a dimension are aligned the same way, except for the inner dimension, equation A.5 verifies this correctness. Equation A.5 computes the alignment of elements $(1, \dots, k+1, \dots, n)$ and $(0, \dots, k-1, \dots, n)$ with the two new sizes for hyperplanes of the k^{th} dimension.

$$
\left.\begin{array}{r}
\sum_{a=k+2}^{n-1}\left(i_a * \prod_{b=k}^{a-1} Dim_k * \prod_{b=0}^{k-1} S_b\right) \\
+i_0 + \sum_{a=1}^{k}\left(i_a * \prod_{b=0}^{a-1} S_b\right) + (2i_{k+1}) * \prod_{b=0}^{k-1} S_b + (i_{k+1}) * S_k^e + (i_{k+1}) * S_k^o \equiv_v c \\
\\
\sum_{a=k+2}^{n-1}\left(i_a * \prod_{b=k}^{a-1} Dim_k * \prod_{b=0}^{k-1} S_b\right) \\
+i_0 + \sum_{a=1}^{k}\left(i_a * \prod_{b=0}^{a-1} S_b\right) + (2i_{k+1}+2) * \prod_{b=0}^{k-1} S_b + (i_{k+1}+1) * S_k^e + (i_{k+1}+1) * S_k^o \equiv_v c
\end{array}\right\} \tag{A.5}
$$

Solving equation A.5 results in having $S_k^o + S_k^e \equiv_v 0$. Or, since $S_k^e \equiv_v 1$ and $S_k^o \equiv_v v-1$, it is always true, hence confirming the correctness of the alignment of the required cells for an update in this Jacobi stencil example with any number of dimensions.

Annexe B

Execution of the MSFT algorithm

Consider the task graph given in Figure B.1. The seven tasks A, B, C, D, E, F & G are depicted as nodes in the DAG (Figure B.1(a)) together with the edges labelled with communication costs. The execution latencies of the tasks on three processors, P_1, P_2 & $P3$ are given in Figure B.1(b).

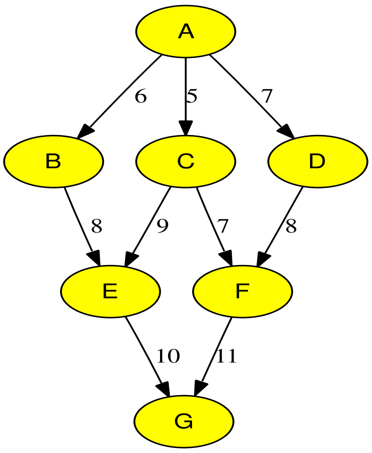
The execution trace of the first phase of MSFT is shown in Figure B.2. The Earliest Execution (EE) time values computed for each node corresponding to the processors P_1, P_2 & $P3$ are given as bottom labels of the nodes. Since the node A has no predecessor, it is assigned the EE values of 7, 9 & 5 for processors P_1, P_2 & $P3$ respectively. For the node B, the EE for the processor P_1 is computed as $\min(7+0+6, 9+6+6, 5+6+6) = 13$. Similarly, for P_2 and P_3, EE for the node B becomes $\min(7+6+8, 9+0+8, 5+6+8) = 17$ and $\min(7+6+10, 9+6+10, 5+0+10) = 15$ respectively. For a node having more than one predecessor, the maximum value corresponding to its predecessors is used for computation of EE values as given in Equation 4.3. For example, for the node E, the EE value for processor P_1 is 21, produced as the maximum of $min(13+0+7, 17+8+7, 15+8+7)$ and $min(14+0+7, 18+9+7, 13+9+7)$. All the subsequent nodes are processed similarly, and the EE values are computed. The priority is then assigned using the smallest EE value[1] produced by any of the processors.

Using the priority criteria, the nodes are processed in the order A, D , B, C, F, E & G. Figures B.3(a), B.3(b), B.4(a) & B.4(b) depict the processor assignments (as node labels) and the SFT values with stride equal to 1 corresponding to each processor (as node bottom labels) for nodes A, D, B & C respectively.

The SFT values for the node A using processors P_1, P_2 & P_3 are $max(0+7+6, 0+7+7, 0+7+5) = 14$, $max(0+9+8, 0+9+9, 0+9+6) = 18$ & $max(0+5+10, 0+5+8, 0+5+7) = 15$ respectively. The processor P_1 producing the smallest SFT value is therefore assigned to the node A as shown in Figure B.3(a). For the node D, the SFT values for the processors P_1, P_2 & P_3 are 18, 28 & 28 respectively. This results in the processor P_1 being assigned to the task D as shown in Figure B.3(b). For the node B, the SFT values are 25, 32 & 29 which result in processor P_1 being assigned to the task as shown in Figure B.4(a). Similarly, for the node C, the processor P_3 is assigned to the task since its assignment produces the minimum SFT value equal to 28 as shown in Figure B.4(b).

The subsequent execution of the algorithm results in the processor P_3 being assigned to the nodes F & G. The final assignments of processors to the tasks are

[1] The smallest EE value is used for priority computation, as with heterogeneous systems, the processor with the smallest EE value is more probable to be assigned.



(a) Task graph having communication costs as labels

Task	**P1**	**P2**	**P3**
A	7	9	5
B	6	8	10
C	7	9	8
D	5	6	7
E	7	9	8
F	6	8	7
G	7	8	6

(b) Task execution costs on processors P_1, P_2 & P_3

FIGURE B.1 – Task graph with communication and execution costs.

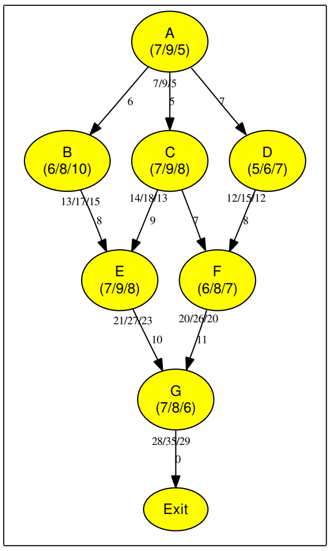


FIGURE B.2 – Computation of the Earliest Execution (EE) times

shown in Figure B.5.

The nodes A, D , B, C, F, E & G have AFT values of 7, 12, 18, 20, 27, 35 & 41 respectively. Consequently, the the MSFT scheduling algorithm produces an optimal schedule of length 41. In contrast, the HEFT algorithm produces a schedule of length 45.

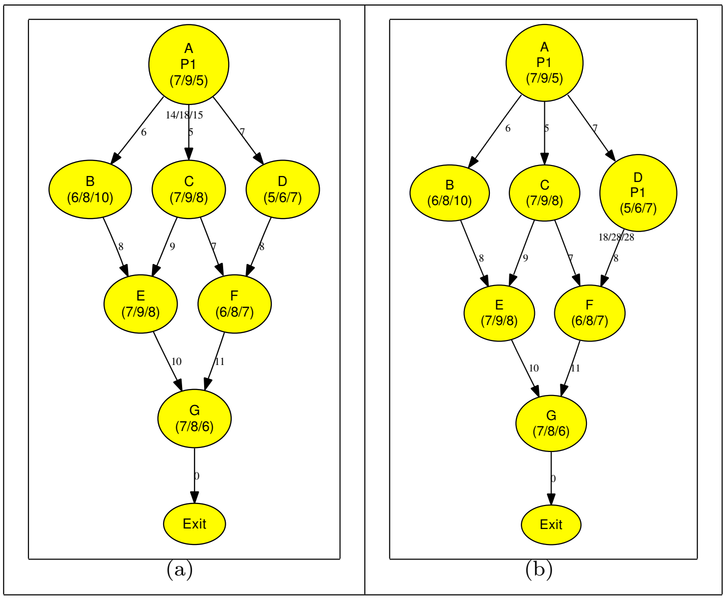


FIGURE B.3 – Processor assignment for the nodes A and D

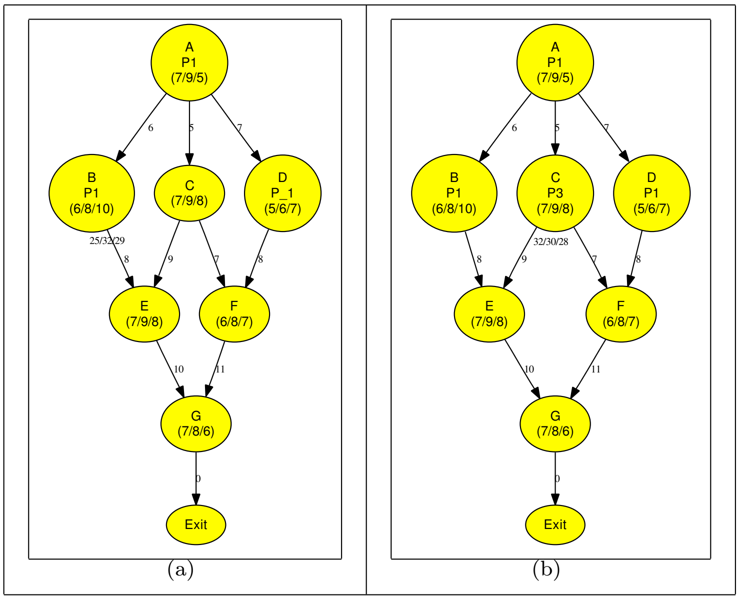


FIGURE B.4 – Processor assignment for the nodes B and C

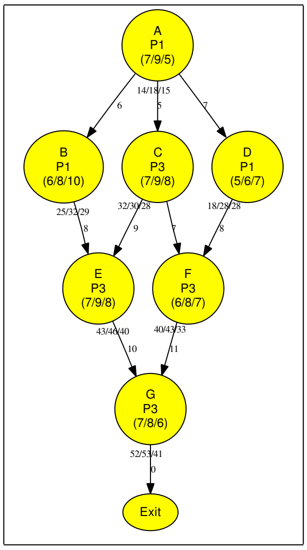


Figure B.5 – Final processor assignments

Bibliographie

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. Oboyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *In Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 295–305. IEEE Computer Society, 2006.

[2] I. Ahmad and Y.-K. Kwok. A new approach to scheduling parallel programs using task duplication. In *International Conference on Parallel Processing*, pages 47–51, NC, USA, August 1994. CRC Press.

[3] M. Al-Mouhamed. Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Transactions on Software Engineering*, 16 :1390–1401, 1990.

[4] R. Allen and K. Kennedy. Automatic loop interchange. *SIGPLAN Not.*, 39(4) :75–90, Apr. 2004.

[5] V. A. F. Almeida, I. M. M. Vasconcelos, J. N. C. Árabe, and D. A. Menascé. Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems. In *Supercomputing '92 : Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 683–691, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215 :403–410, 1990.

[7] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loop. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'91)*, pages 39–50, June 1991.

[8] I. Anghel, T. Cioara, I. Salomie, G. Copil, D. Moldovan, and C. Pop. Dynamic frequency scaling algorithms for improving the cpu's energy efficiency. In *Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference*, pages 485–491. IEEE, 2011.

[9] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory : design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[10] D. Barthou, S. Donadio, A. Duchateau, P. Carribault, and W. Jalby. Loop optimization using adaptive compilation and kernel decomposition. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, pages 170–184, San Jose, California, Mar. 2007. IEEE Computer Society.

[11] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, Reading, MA, 1994.

[12] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Intl. Conference on Compiler Construction*, pages 244–263, 2010.

[13] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, O. Temam, A. Group, and I. Rocquencourt. Putting polyhedral loop transformations to work. In *In Workshop on Languages and Compilers for Parallel Computing (LCPC'03), LNCS*, pages 209–225, 2003.

[14] F. Belletti, S. F. Schifano, R. Tripiccione, F. Bodin, P. Boucaud, J. Micheli, O. Pene, N. Cabibbo, S. de Luca, A. Lonardo, D. Rossetti, P. Vicini, M. Lukyanov, L. Morin, N. Paschedag, H. Simma, V. Morenas, D. Pleiter, and F. Rapuano. Computing for LQCD : apeNEXT. *Computing in Science and Engineering*, 8(1) :18–29, 2006.

[15] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the intelÂ® architecture. *International Journal of Parallel Programming*, 30 :65–98, 2002. 10.1023/A :1014230429447.

[16] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2) :135–151, June 2002.

[17] P. Boyle, D. Chen, N. Christ, C. Cristian, Z. Dong, A. Gara, B. Joï¿$\frac{1}{2}$, C. Kim, L. Levkova, X. Liao, G. Liu, R. Mawhinney, S. Ohta, T. Wettig, and A. Yamaguchi. Status of the QCDOC project. *arXiv :hep-lat/0110124v1*, 2001.

[18] P. A. Boyle, D. Chen, N. H. Christ, M. A. Clark, S. D. Cohen, C. Cristian, Z. Dong, A. Gara, B. Joo, C. Jung, C. Kim, L. A. Levkova, X. Liao, R. D. Mawhinney, S. Ohta, K. Petrov, T. Wettig, and A. Yamaguchi. Overview of the QCDSP and QCDOC Computers. *IBM Journal of Research and Development*, 49(2-3) :351–366, 2005.

[19] D. Bozda ;, F. Ozguner, and U. V. Catalyurek. Compaction of schedules and a two-stage approach for duplication-based dag scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 20 :857–871, 2009.

[20] H. C. Brearley. Illiac ii-a short description and annotated bibliography. *Electronic Computers, IEEE Transactions on*, EC-14(3) :399 –403, june 1965.

[21] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *In Proceedings of the 29th Annual Hawaii International Conference on System Sciences, Maui, HI*, pages 183–192, 1996.

[22] P. Carribault, A. Cohen, and W. Jalby. Deep jam : Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *IEEE PACT*, pages 291–302, 2005.

[23] H. B. Chen, B. Shirazi, K. Kavi, and A. R. Hurson. Static scheduling using linear clustering with task duplication. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing and Systems*, pages 285–290, 1993.

[24] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB*, 1(2) :1313–1324, 2008.

[25] J. chiou Liou and M. A. Palis. An efficient task clustering heuristic for scheduling dags on multiprocessors. In *Multiprocessors, Workshop on Resource Management, Symposium of Parallel and Distributed Processing*, pages 152–156, 1996.

[26] B. Cirou and E. Jeannot. Triplet : a clustering scheduling algorithm for heterogeneous systems. In *IEEE Symposium on Reliable Distributed Systems*, pages 231–236. IEEE, 2001.

[27] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 151–160, New York, NY, USA, 2005. ACM.

[28] C. J. Conti, D. H. Gibson, and S. H. Pitkowsky. Structural aspects of the system/360 model 85, i : General organization. *IBM Systems Journal*, 7(1) :2 –14, 1968.

[29] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, 1999.

[30] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *In Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9. ACM Press, 1999.

[31] N. Corporation. Cuda programming guide. 2010.

[32] K. Czarnecki, J. T. Odonnell, J. Striegnitz, Walid, and Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. *LNCS : Domain-Specific Program Generation*, 3016(2) :51–72, 2004.

[33] S. Darbha and D. Agrawal. Sdbs : a task duplication based optimal scheduling algorithm. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 756 –763, 23-25 1994.

[34] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1) :129–159, 2009.

[35] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *ACM Intl. Conference on Supercomputing*, page 4, Austin, Texas, Nov. 2008.

[36] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. Maqao : Modular assembler quality analyzer and optimizer for itanium 2. Mar. 2005.

[37] A. Doan and Fsunzgner. Ldbs : A duplication based scheduling algorithm for heterogeneous computing systems. *Parallel Processing, International Conference on*, 0 :352, 2002.

[38] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Phys. Lett.*, B195 :216–222, 1987.

[39] S. W. Dunwell. Design objectives for the ibm stretch computer. In *Papers and discussions presented at the December 10-12, 1956, eastern joint computer conference : New developments in computers*, AIEE-IRE '56 (Eastern), pages 20–22, New York, NY, USA, 1957. ACM.

[40] H. Dursun, K. ichi Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par*, pages 642–653, 2009.

[41] H. Dursun, K. ichi Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, 2009.

[42] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, pages 82–93, 2004.

[43] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.*, 9(2) :138–153, 1990.

[44] J. R. Ellis. *Bulldog : a compiler for VLSI architectures.* MIT Press, Cambridge, MA, USA, 1986.

[45] T. M. B. K. Eric Petit1, Pablo de Oliveira Castro and W. Jalby. Computing-kernels performance prediction using data flow analysis and microbenchmarking. In *CPC*, 2011.

[46] L. Fireman, E. Petrank, and A. Zaks. New algorithms for simd alignment. In *Intl. Conference on Compiler Construction*, pages 1–15, Braga, Portugal, Mar. 2007.

[47] M. Frigo and S. Johnson. Fftw : an adaptive software architecture for the fft. In *Intl. Conference on Acoustics, Speech and Signal Processing.*, volume 3, pages 1381 –1384 vol.3, may 1998.

[48] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *High Performance Enmebdded Architectures and Compilers*, pages 29–46. Springer Berlin / Heidelberg, 2005.

[49] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4) :276 – 291, 1992.

[50] S. Girbal. *Optimisation d'applications - Composition de transformations de programme : modÃ¨le et outils.* PhD thesis, University Paris-Sud 11, Orsay, France, September 2005.

[51] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3) :261–317, 2006.

[52] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Intl. Conference on Compiler Construction*, pages 225–245, Saarbrücken, Germany, Mar. 2011.

[53] IBM. Exploiting the dual floating point units in blue gene/l. IBM developper support.

[54] J. Jaeger and D. Barthou. Combining experimental search and performance model for adaptive optimization. In *Hipeac Workshop on Statistical and Machine Learning Approaches to Architectures and compilation*, Paphos, Cyprus, Jan. 2009.

[55] M. Joyce, Thomas F. (Burlington. Round robin replacement for a cache store, March 1980.

[56] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IEEE Intl. Parallel and Distributed Processing Symposium*, pages 1–12, Atlanta, Georgia, Apr. 2010.

[57] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick. Implicit and explicit optimizations for stencil computations. In *Workshop on Memory System Performance and Correctness*, pages 51–60, San Jose, California, Oct. 2006.

[58] S. J. Kim and J. C. Brown. A general approach to mapping of parallel computations upon multiprocessor architectures. In *in Proceedings of the International Conference on Parallel Processing*, pages 1–8, 1988.

[59] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00 : Proceedings of the 2000 International Conference on Parallel Architectures*

and Compilation Techniques, Washington, DC, USA, 2000. IEEE Computer Society.

[60] S. Krishnamoorthy, M. M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 235–244, San Diego, California, June 2007.

[61] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software*, 5 :23–32, 1988.

[62] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI'04, proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 171–182, 2004.

[63] Y.-K. Kwok and I. Ahmad. Exploiting duplication to minimize the execution times of parallel programs on message-passing systems. In *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*, pages 426 –433, 26-29 1994.

[64] Y. kwong Kwok, I. Ahmad, and I. Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7 :506–521, 1996.

[65] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. D. Anger. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, 7(3) :141 – 147, 1988.

[66] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth : an evaluation of throughput computing on cpu and gpu. pages 451–460, Saint Malo, France, June 2010. ACM Press.

[67] Y.-J. Lee and M. Hall. A code isolator : Isolating code fragments from large programs. pages 164–178, 2005.

[68] L. Li, L. Gao, and J. Xue. Memory coloring : a compiler approach for scratchpad memory management. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 329 – 338, sept. 2005.

[69] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Trans. on Programming Languages and Systems*, 26 :2004, 2004.

[70] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T. fook Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 348–357, Raleigh, North Carolina, Sept. 2009.

[71] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. *Heterogeneous Computing Workshop*, 0 :57, 1998.

[72] S. Naci. Optimizing inter-nest data locality using loop splitting and reordering. In *IPDPS*, pages 1–8. IEEE, 2007.

[73] A. D. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *SC*, pages 1–13, New Orleans, LA, Nov. 2010. IEEE Computer Society.

[74] V. H. Nguyen and D. Lavenier. Speeding up subset seed algorithm for intensive protein sequence comparison. pages 57–63, 2008.

[75] N.Thomas, G.Tanase, O.Tkachyshyn, J.Perdue, N.Amato, and L.Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Proc. ACM PPOPP'05*, Chicago, 2005.

[76] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor simd : Auto-vectorize once, run everywhere. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 151–160, Chamonix, France, 2011.

[77] D. Nuzman, M. Namolaru, A. Zaks, and J. H. Derby. Compiling for an indirect vector register architecture. In *ACM Computing Frontiers Conf.*, pages 199–208, Ischia, Italy, May 2008. ACM Press.

[78] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 132–143, Ottawa, Ontario, June 2006.

[79] D. Nuzman and A. Zaks. Outer-loop vectorization : revisited for short simd architectures. In *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 2–11, Toronto, Ontario, Oct. 2008. ACM Press.

[80] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12) :1184–1201, Dec. 1986.

[81] G. Park, B. Shirazi, and J. Marquis. Dfrn : A new approach for duplication based scheduling for distributed memory multiprocessor systems. *Parallel Processing Symposium, International*, 0 :157, 1997.

[82] L. Plagne and F. Hülsemann. Improving Large Vector Operations with C++ Expression Template and ATLAS, 2007. MPOOL07 web page :
http ://homepages.fh-regensburg.de/ mpool/mpool07/proceedings/11.pdf.

[83] L. Plagne and F. Hülsemann. BTL++ : From Performance Assessment to Optimal Libraries. In M. Bubak, G. D. van Albada, P. J. J. Dongarra, and M. Sloot, editors, *Proceedings of the 8th International Conference on Computational Science (ICCS'08), Part III*, volume 5103 of *LNCS*, pages 203–212, Kraków, Poland, June 2008. Springer-Verlag.

[84] L. Plagne, F. Hulsemann, D. Barthou, and J. Jaeger. Parallel expression template for large vectors. In *Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, pages 1—8, Genova, Italy, 2009. ACM Press.

[85] M. Poletto, W. C. Hsieh, D. R. Engler, and F. M. Kaashoek. 'C and tcc : A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21 :324–369, March 1999.

[86] C. J. Purcell. The control data star-100 : performance measurements. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*, AFIPS '74, pages 385–387, New York, NY, USA, 1974. ACM.

[87] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral : Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2) :232 –275, feb. 2005.

[88] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5) :469–498, Oct. 2000.

[89] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *ACM Intl. Conference on Supercomputing*, 2000.

[90] R. M. Russell. The cray-1 computer system. *Commun. ACM*, 21 :63–72, January 1978.

[91] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2) :175–187, 1993.

[92] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 409 –415, 2002.

[93] V. Sundriyal, M. Sosonkina, F. Liu, and M. W. Schmidt. Dynamic frequency scaling and energy saving in quantum chemistry applications. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, pages 837–845. IEEE, 2011.

[94] F. Suter, F. Desprez, and H. Casanova. From heterogeneous task scheduling to heterogeneous mixed parallel scheduling. In *In Euro-Par*, pages 230–237. Vivien, 2004.

[95] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. pages 117–128, 2011.

[96] J. E. Thornton. *Design of a Computer&#8212;The Control Data 6600*. Scott Foresman & Co, 1970.

[97] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. *Heterogeneous Computing Workshop*, 0 :3, 1999.

[98] S.-A.-A. Touati and D. Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 147–156, New York, NY, USA, 2006. ACM.

[99] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *CoRR*, abs/1004.1741, 2010.

[100] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 327–337, Raleigh, North Carolina, Sept. 2009.

[101] C. Urbach. Lattice QCD with Two Light Wilson Quarks and Maximal Twist. *The XXV International Symposium on Lattice Field Theory*, 2007.

[102] C. Urbach, K. Jansen, A. Shindler, and U. Wenger. HMC Algorithm with Multiple Time Scale Integration and Mass Preconditioning. *Computer Physics Communications*, 174 :87, 2006.

[103] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley, BostonMA, MA USA, 2002.

[104] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5) :26–31, 1995.

[105] T. L. Veldhuizen. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[106] S. Venkatasubramanian and R. W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *ACM Intl. Conference on Supercomputing*, pages 244–255, Yorktown Heights, NY, June 2009. ACM Press.

[107] P. Vranas, M. A. Blumrich, D. Chen, A. Gara, M. E. Giampapa, P. Heidelberger, V. Salapura, J. C. Sexton, R. Soltz, and G. Bhanot. Massively Parallel Quantum Chromodynamics. *IBM journal of research and development*, 52(1/2), 2008.

[108] J. Walter and M. Koch. uBLAS web page :
http ://www.boost.org/libs/numeric/ublas.

[109] L. Wang, M. Huang, and T. El-Ghazawi. Towards efficient gpu sharing on multicore processors. In *Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, PMBS '11, pages 23–24, New York, NY, USA, 2011. ACM.

[110] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(2) :234–248, 2001.

[111] M. Weiss. Strip mining on simd architectures. In *Proceedings of the 5th international conference on Supercomputing*, ICS '91, pages 234–243, New York, NY, USA, 1991. ACM.

[112] R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Softw. Pract. Exper.*, 38(15) :1621–1642, 2008.

[113] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software : Practice and Experience*, 35(2) :101–121, February 2005. "ATLAS web page : http ://math-atlas.sourceforge.net".

[114] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. 27(1-2) :3–35, 2001.

[115] M. Wolfe. Loops skewing : The wavefront method revisited. *International Journal of Parallel Programming*, 15 :279–293, 1986. 10.1007/BF01407876.

[116] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, USA, 1990.

[117] D. Wonnacott. Achieving scalable locality with time skewing. *Intl. J. of Parallel Programming*, 30(3) :1–221, 2002.

[118] W. A. Wulf and S. A. Mckee. Hitting the memory wall : Implications of the obvious. *Computer Architecture News*, 23 :20–24, 1995.

[119] T. Yang and A. Gerasoulis. Dsc : Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5 :951–967, 1994.

[120] K. Yotov, X. Li, G. Ren, M. Garzarán, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High-Performance BLASs ? *Proc. of the IEEE*, 93(2) :358–386, February 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[121] M. you Wu and D. D. Gajski. Hypertool : A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1 :330–343, 1990.

[122] C.-K. L. Yuan Tang, Rezaul Alam Chowdhury and C. E. Leiserson. Coding Stencil Computation using the Pochoir stencil-specification language. In *Hot-Par '11 : 3rd USENIX Workshop on Hot Topics in Parallelism*, Berkeley, California, May 2011.

[123] R. G. Zwakenberg. Cdc 6600/7600 optimization. *SIGPLAN Not.*, 5 :130–, July 1970.