Implementation trade-offs for FGPA accelerators

Gaël Deest

ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MathSTIC

présentée par

Gaël Deest

préparée à l'unité de recherche 6074 - IRISA
Institut de recherche en informatique et systèmes Aléatoires
UFR Informatique Électronique

Implementation Trade-Offs for FPGA Accelerators

Thèse soutenue à Rennes le 14 décembre 2017

devant le jury composé de :

Albert COHEN
DR Inria Rocquencourt / rapporteur

Florent DE DINECHIN
PR INSA Lyon / rapporteur

Karine HEYDEMANN
MCF Université Pierre et Marie Curie / examinatrice

Daniel MÉNARD
PR INSA Rennes / examinateur

Tomofumi YUKI
CR Inria Rennes / examinateur

Steven DERRIEN
PR Université de Rennes 1 / directeur de thèse

Remerciements

Chaque thèse est une expérience unique. Si, pour certains, il semble ne s'agir que d'un rite de passage sans réelle difficulté, tel ne fut assurément pas mon cas. Au cours de ces ~~trois~~ quatre années éprouvantes, ponctuées de doutes et de joies, d'intuitions et d'espoirs déçus, j'ai eu la chance d'être entouré de gens formidables, sans l'appui desquels ce manuscrit n'existerait sans doute pas.

En premier lieu, je remercie mes directeurs de thèse, Steven Derrien et Olivier Sentieys, d'avoir accordé leur confiance au vieil étudiant que j'étais et de m'avoir ouvert les portes de l'équipe Cairn, d'abord comme stagiaire, puis comme doctorant. En outre et en particulier, je remercie Steven pour sa tutelle scientifique exigeante, parfois dure mais souvent juste – si j'ai un tant soit peu appris à ravaler mon orgueil, me poser les bonnes questions et raisonner en chercheur, c'est en grande partie grâce à lui.

On ne saurait trop souligner l'apport de Tomofumi Yuki à ces travaux. Si, décidément, mon esprit trop français a bien du mal à appréhender le concept de *punchline*, son attachement à me l'expliquer encore et encore illustre bien sa considérable patience. Je le remercie encore pour sa rigueur, sa bienveillance et son humilité.

Je remercie le jury d'avoir accepté la charge d'évaluer ces travaux. La pertinence de leurs remarques et questions prouve qu'ils se sont acquittés de cette tâche avec un sérieux remarquable – j'en suis profondément honoré. En particulier, j'adresse de profonds remerciements à mes deux rapporteurs, Albert Cohen et Florent de Dinechin, pour la lecture approfondie qu'ils ont faite de ce manuscrit et leur regard expert sur mes contributions.

Je remercie tous les membres de l'équipe Cairn, permanents, post-docs, doctorants et stagiaires, présents et passés, de m'avoir si bien accueilli parmi eux. Que serait une journée au labo sans les deux pauses réglementaires et les débats plus ou moins trollesques qui les animent ? En vrac, et de façon non exhaustive, je remercie chaleureusement Ali Hassan El Moussawi, Antoine Morvan, Nicolas Simon, Christophe Huriaux, Laurent Perraudeau, Angeliki Kritikakou, Patrice Quinton, Nicolas Estibals, Baptiste Roux, Simon Rokicki, Thomas Levesque, Rafail Psiakis...

Merci à notre assistante, Nadia Derouault, pour sa gentillesse, son humanité, sa prévenance et son indulgence quant à ma totale incompétence administrative.

Merci aussi à mes anciens comparses de Master rennais, devenus de précieux amis, tant pour les discussions enflammées sur IRC que leur soutien moral: Simon Bouget, Gabriel Radanne, Benoit Le Gouis et Nicolas Braud-Santoni.

Je remercie également mon ami de quinze ans, Jérémy Hervé, tant pour son affection et son humour décapant que pour avoir été là dans les moments difficiles. Sans doute ignorais-tu, en me parlant de FPGA en 2002, que je finirais par faire une

thèse sur le sujet, après un chemin bien sinueux !

Merci à ma famille pour tout ce qu'elle m'a enseigné, malgré les épreuves et les tempêtes.

Merci enfin à celle dont je partage la vie depuis plus de onze années et qui a cru en moi comme personne, vivant ces dernières années avec la même intensité que moi: Caroline, mon Amour.

Contents

List of Figures

List of Tables

Résumé en français

0.1 Contexte

Des téléphones aux centres de calcul, les systèmes informatiques jouent aujourd'hui un rôle majeur dans la plupart des activités humaines. On les retrouve dans des contextes variés, dont la diversité reflète celle de leurs applications. En première approche, on peut distinguer trois types de système informatique:

Les systèmes généralistes, comme les stations de travail, sont conçus pour effectuer raisonnablement efficacement un bon nombre de tâches (comme naviguer sur le web, utiliser une suite bureautique ou jouer à des jeux vidéos). Leur principale caractéristique est leur flexibilité, puisqu'ils peuvent exécuter des programmes arbitraires installés ou écrits par l'utilisateur final.

Les systèmes embarqués, en revanche, sont spécifiquement dédiés à une ou quelques tâches bien spécifiques. Ils sont enfouis dans toutes sortes de système, des smartphones (où ils accélèrent par exemple les opérations d'encodage/décodage) au module de commande des engins spatiaux. Ils se caractérisent par les contraintes fortes qui pèsent sur leur conception. En effet, ils doivent par exemple respecter des contraintes de latence (temps réel), de consommation énergétique, de taille, de coût et de tolérance aux fautes – souvent simultanément.

Les super-ordinateurs, par contraste, sont optimisés pour des tâches de calcul intensif, comme les simulations scientifiques (par exemple en climatologie ou météorologie) et l'intelligence artificielle (réseaux de neurones). Par rapport aux systèmes généralistes, la différence majeure est leur grande puissance de calcul, puisqu'ils doivent effectuer un grand nombre d'opérations par seconde pour satisfaire aux exigences de performance.

Toutes ces différences influent sur la façon dont ces systèmes sont conçus et programmés. Par exemple, afin de supporter un grand nombre de cas d'utilisation, les ordinateurs généralistes sont basés sur des processeurs génériques offrant un jeu d'instruction prédéfini. Afin d'améliorer les performances, plusieurs stratégies sont mises en oeuvre dans ces architectures, comme l'utilisation de plusieurs niveaux de cache ou de techniques *superscalaires*. Ainsi, un processeur moderne peut, par exemple, ordonner les instructions dynamiquement en fonction des ressources disponibles, ou "deviner" la valeur d'une condition de branchement à partir du passé afin d'exécuter certaines parties du code par avance. Ces mécanismes complexes profitent à la fois aux programmeurs et aux utilisateurs finaux:

- Le même jeu d'instruction peut être réutilisé d'une génération de processeur à l'autre, fournissant une inter-compatibilité ascendante et descendante entre logiciel et matériel.
- Les programmes peuvent être écrits dans un langage de programmation haut niveau, traduit ensuite en une série d'instruction élémentaires par un compilateur, sans qu'il soit nécessaire de connaître précisément l'architecture sous-jacente.
- Les techniques d'optimisation statiques (à la compilation) et dynamiques (mises en oeuvre lors de l'exécution, comme les techniques superscalaires) assurent dans la plupart des cas des performances décentes aux utilisateurs, sans efforts excessifs de la part des programmeurs.

Ces commodités ont rendu possible les environnements logiciels sophisitiqués que nous utilisons quotidiennement. Malheureusement, ces avantages ne sont pas gratuits: par nature, les architectures génériques ne peuvent fournir des performances ou une efficacité optimales, quelle que soit l'application.

En fait, les processeurs génériques représentent un compromis pertinent entre performance et flexibilité. Si ce compromis convient à beaucoup d'applications, tel n'est pas toujours le cas. Par exemple, dans le cadre de systèmes embarqués hautement contraints en ressources, l'utilisation de processeurs génériques peut entraîner un dépassement du budget alloué en termes de surface de silicium ou de consommation énergétique. L'utilisation d'architectures spécifiques, nommées *accélérateurs matériel*, s'impose alors.

Cette thèse traite la conception de tels d'accélérateurs (plus spécifiquement d'accélerateurs implémentés sur FPGA). Nous nous intéressons à la conception de modèles de performance permettant la mise en oeuvre de compromis spécifiques à chaque application, selon diverses métriques (précision, surface, débit, etc.). Ce manuscrit est composé de deux parties majeures: dans les chapitres 2 et 3, l'on s'intéresse aux compromis entre précision et coût matériel (notamment). Dans les chapitres 4 et 5, on se concentre sur une classe d'applications, les *stencils*. Le reste de ce chapitre présente succintement ces deux problématiques et nos contributions.

0.2 Compromis entre coût et précision

Dans la première moitié de ce manuscrit, on s'intéresse à la précision des résultats. Les compromis portant sur la précision représentent un vaste champ d'opportunités pour les architectes matériel. Un exemple classique est l'utilisation de l'arithmétique en virgule fixe au lieu de l'arithmétique en virgule flottante pour réduire les coûts matériel et la consommation énergétique. Naturellement, la précision ne peut être réduite indéfiniment: chaque implémentation doit satisfaire à des contraintes de précision spécifiques, dont le respect doit être vérifié lors de l'exploration de l'espace de conception.

Déterminer si une implémentation en virgule fixe donnée respecte une contrainte de précision représente un problème difficile en général. On peut distinguer deux classes de techniques: simulatoires et analytiques. Les techniques basées sur la

simulation sont facilement applicables et offrent de bons résultats à condition de disposer de suffisamment d'échantillons. Cependant, elles sont lentes à mettre en oeuvre car estimer la précision de chaque solution requiert un grand nombre d'exécutions. Les modèles analytiques sont beaucoup plus rapides à évaluer, ce qui permet l'exploration de plus de solutions en peu de temps, et donc l'identification de meilleur compromis. Cependant, leur applicabilité limitée représente un défi majeur.

Avant ces travaux, les techniques analytiques ne pouvaient traiter que des systèmes uni-dimensionnels. dans le chapitre 3, nous étendons les techniques précédentes à des algorithmes multi-dimensionnels, comme des filtres d'image. Nous nous concentrons sur les filtres Linéaires, Spatialement Invariants (LSI), une généralisation des filtres Linéaires, Invariants dans le Temps supportés par d'autres approches. Nous proposons un flot partant d'une description algorithmique (écrite en C/C++). Les deux principaux défis que nous relevons sont:

- Extraire une représentation mathématique compacte d'un filtre linéaire à partir d'une description impérative en C/C++.
- Dériver un modèle de précision fiable à partir d'une telle représentation.

Le premier de ces défis est relevé dans le cadre du modèle polyédrique. Nous représentons les filtres LSI comme des Systèmes d'Équations aux Récurrences Uniformes (SUREs) ou, de façon équivalente, des graphes de flots de donnée multi-dimensionnels (MDFGs), par analogie aux graphes de flot de signal (SFGs) utilisés comme représentation intermédiaire par les approches précedentes.

Une différence majeure entre SFGs et MDFGs est que les MDFGs / SUREs n'imposent pas d'ordre d'itération canonique à chaque dimension. Cela nous permet de supporter des filtres d'image récursifs complexes, scannant leurs entrées dans toutes les directions. Nous utilisons des techniques polyédriques d'analyse de dépendance afin de transformer le programme en systèmes d'équation aux récurrences *affines*. Un certain nombre de simplifications et de transformations sont requises, comme l'uniformisation des dépendances, avant que le système puisse être reconnu comme un SURE.

Pour la seconde problématique – inférer des modèles de précision – nous proposons deux approches. Toutes deux se ramènent à calculer l'intégrale et la norme L^2 de la réponse impulsionnelle du filtres, mais depuis des points de vue duaux:

- Dans le domaine temporel, nous dérivions ces sommes en déroulant / évaluant les équations de récurrence définissant le système.
- Dans le domaine fréquentiel, nous exploitons les propriétés algébriques des fonctions de transfert pour calculer celles représentant la propagation de chaque erreur. Nous calculons alors les sommes requises à partir de la réponse fréquentielle.

Pour le deuxième cas, nous proposons une version simplifiée et plus efficace de l'algorithme proposé par Ménard et al. [23]. Nos expériences démontrent que nos modèles sont obtenus rapidement, et leur efficacité est illustrée en comparant avec des simulations sur des données réelles.

Finalement, nous montrons comment l'approche fréquentielle peut être utilisée en amont de l'optimisation des largeurs afin de traiter la quantification des coefficients, un problème généralement ignoré dans les autres travaux.

0.3 Compromis pour l'implémentation de stencils

Dans la seconde partie de cette thèse, nous nous concentrons sur les compromis possibles pour l'implémentation de stencils itératifs sur FPGA. Les stencils itératifs (ou plus simplement stencils) sont un motif de calcul retrouvé dans de nombreuses applications, des simulations scientifiques à la vision par ordinateur. Chaque application présente des contraintes spécifiques en fonction de la taille du domaine, du schéma de dépendance et des caractéristiques intrinsèques du calcul.

En première approche, les performances d'un stencil sont principalement déterminées par les ressources de calcul utilisées et les performances de la mémoire. Le *pavage* (ou *tiling*), présenté dans le chapitre 4, est un outil essentiel pour optimiser ces deux aspects, en améliorant la localité mémoire d'une part et en autorisant la parallélisation à différentes niveaux d'autre part.

Au chapitre 5, nous proposons une méthode systématique pour l'impl'ementation de stencil itératifs sur FPGA. Notre méthode s'appuie sur un gabarit d'architecture flexible, fondé sur le pavage et exposant diverses paramètres:

- Les performances maximales peuvent être controlées en ajustant le facteur de déroulage du chemin de données. Ceci autorise des compromis entre débit et surface.
- Des tuiles plus grandes peuvent être utilisées pour réduire les besoins en bande passante, au prix d'une plus grande utilisastion en mémoire locale.
- L'espace d'itération peut être pavé selon un sous-ensemble de ses dimensions, afin de réduire encore plus l'utilisation en bande passante. Le prix à payer est une perte de contrôle partielle sur l'utilisation en mémoire locale, qui devient proportionnelle à la taille de chaque dimension non pavée.

En outre, nous proposons des modèles de performance simples, dérivés d'une analyse à haut niveau des performances du système. Ces modèles peuvent servir de base à l'exploration de l'espace des solutions.

Pour valider notre architecture et nos modèles, nous avons implémentńotre approche sous la forme d'un outil de génération de code visant Vivado SDSoC. Nous avons identifié, pour différentes cibles de performance, plusieurs solutions potentielles en utilisant nos modèles de performance / surface. Nos expériences démontrent la bonne précision de nos modèles de performance. Nos modèles de surface s'avèrent, bien que moins précis, suffisants pour estimer a priori les solutions les plus intéressantes. Ces modèles ont donc fait la preuve de leur utilité lors de la phase de conception.

Finalement, au cours de ce travail, nous avons constaté l'importance de la contiguité en mémoire pour réduire la latence et bénéficier des accès *bursts* proposés par le bus. Nous avons donc conçu une disposition mémoire spécifique pour les tencils.

Pour l'instant, ce layout n'est implémenté que pour les stencils 2D. Sa généralisation à des stencils de dimension supérieure mériterait d'autres travaux.

0.4 Perspectives

Nos travaux ouvrent plusieurs perspectives.

Une direction de recherche évidente consisterait à étendre notre travail sur les modèles de précision à une classe plus grande de programmes, comme les filtres linéaires non-invariants, ou des filtres constitués d'opérations arbitraires. Une autre piste, peut-être plus intéressante, serait de supporter des programmes non polyédriques. Par exemple, certains algorithmes, comme la transformée de Fourier rapide, présentent de grandes régularités et un flot de contrôle statique sans pour autant être représentables avec des dépendances affines. La propagation des erreurs dans de tels algorithmes présente encore des défis, et nous ne savons pas si le déroulage peut être évité.

On pourrait aussi imaginer utiliser des modèles de précision analytiques dans d'autres contextes, par exemple pour analyser les erreurs de quantification dans des programmes en virgule flottante, ou pour prédire l'impact d'erreurs transitoires ("soft errors") ou d'opérateurs approximatifs sur la correction du programme. Une difficulté majeure est que, dans de tels cas, les erreurs ne vérifient pas les mêmes propriétés statistiques que le bruit de quantification de le cas de la virgule fixe.

Nos travaux sur les stencils s'appuie sur une compréhension claire des facteurs majeurs affectant leur performance. Des observations similaires peuvent être formul'ees pour de nombreux algorithmes. Une approche plus générique, ciblant tous les algorithmes se prêtant au pavage, pourrait probablement être proposée.

Nos recherches sur le placement contigu des données en mémoire. Cette problématique, importante en pratique, n'a pas été très étudiée. Elle ouvre des questions intéressantes, comme sur la façon de réduire le nombre de tampons requis ou la nécessité d'utiliser des mémoires locales pour réordonner les entrées. Nous suspectons l'existence de compromis intéressants entre la séquentialité des données d'une part et leur contiguïté d'autre part. Cependant, cette question mériterait de plus amples investigations.

Pour finir, l'interaction entre les stencils et la précision est difficile à étudier. La capacité des FPGAs à gérer des données de taille arbitraire constitue l'un de leurs avantages majeurs, autorisant des réductions en surface significatives pour les applications supportant une certaine dégradation de la précision. Pouvoir caractériser l'impact (par exemple) de formats en virgule fixe sur la précision de stencils autoriserait des compromis très intéressants.

0.5 Conclusion

Lors de la conception d'accélerateurs matériel, le défi majeur réside dans la taille de l'espace de conception à explorer, en particulier quand certaines dimensions de design comme la précision, sont envisagées. Comme chaque application possède des exigences spécifiques, une solution unique ne peut répondre à tous les besoins. Dans

cette thèse, nous défendons une approche rigoureuse de la conception matérielle, basée sur l'utilisation de modèles. Nous avons démontré l'efficacité de cette stratégie à deux problématiques distinctes: les stencils et l'optimisation des largeurs. À mesure que les accélérateurs se répandent, l'utilisation de modèles spécifiques deviendra essentiel pour comprendre l'impact des choix de conception sur le comportemet du système. Notre travail représente une étape dans cette direction.

Chapter 1

Introduction

1.1 Context

From smartphones to data centers, computer systems now play a major role in most human activities. They are found in vastly different contexts, reflecting the diversity of their applications. We distinguish three main types of computing environments:

General-purpose computers, such as desktop workstations, are designed to run reasonably efficiently a number of applications (e.g., web browsers, office suites or video games). Their main feature is their flexibility, as they can run arbitrary programs installed or written by the end user.

Embedded systems, on the other hand, are dedicated to some specific task or set of tasks. They can be found within all sorts of systems and devices, from smartphones (where they typically handle encoding/decoding tasks) to the command system of spacecrafts. Their characteristic is the highly constrained environment in which they operate, as they are usually subject to real-time, power, size, cost or fault-tolerance constraints (often at the same time).

Supercomputers are optimized towards High-Performance Computing (HPC) workloads, such as scientific simulations (e.g., climatology, seismology) or machine learning applications. Compared to general-purpose computers, the major difference is their significant computing power, as they must perform a large number of operations per second to meet performance requirements.

All these differences influence the way these systems are designed and programmed. For example, since they must support a large number of use cases, general-purpose computers are based on generic processor design offering a predefined instruction set. Several design strategies are used to improve performance, such as adding multiple levels of cache or applying *superscalar* techniques in the processor. A modern processor may thus, for instance, schedule instructions dynamically based on available resources, or "guess" the value of a branching condition based on past executions. These complex mechanisms benefits both end users and programmers, since:

- The same instruction set may be used over several CPU generations, providing backward and forward compatibility between software and hardware.

- Compilers can translate programs written in a high-level programming language into sequences of elementary instructions, without exact knowledge of the supporting architecture.
- Compile- and run-time optimizations (from memory hierarchy to superscalar techniques) ensure that users get decent performance in most cases without excessive optimization efforts from the programmer.

These facilities have made possible the sophisticated software environments that we use daily. Unfortunately, these advantages come at a price: generic architectures cannot, by their very nature, provide optimal performance or efficiency for any particular application.

Generic processors represent a convenient trade-off between performance and flexibility. While this is suitable for many applications, such is not always the case. For instance, in resource-constrained embedded systems, generic processors may exceed power and area budget for a given performance goal. The use of more efficient, special-purpose *hardware accelerators* is then necessary. Such accelerators, and the the problem of their design, are at the core of this thesis.

1.2 Hardware Accelerators

In a broad sense, the term *accelerator* denotes any processing device that gives up *some* genericity to execute a type of computation more efficiently than a general-purpose processor (along which they are commonly used). Floating-point coprocessors, such as Intel's C8087 (introduced in 1980) constitute good examples[1]. More generally, for the purpose of this discussion, we distinguish:

- Programmable accelerators, designed for a class of applications while retaining some level of programmability. Well-known examples include Digital Signal Processors (DSPs) and Graphics Processing Units (GPUs). DSPs are special-purpose processors that offer efficient support for common signal processing operations (e.g., dot products). GPUs, on the other hand, have evolved from domain-specific chips into powerful semi-generic computing platforms for data-parallel floating-point computations.
- Custom, fixed-function accelerators, on the other hand, are specifically designed for a single, well-defined task. They may be implemented as costly Application-Specific Integrated Circuits (ASICs), or on top of reconfigurable logic, such as Field-Programmable Gate Arrays (FPGAs).

Naturally, fixed-function implementations represent the highest level of specialization, and offer the greatest potential of optimizations. Notice, though, that such architectures move the responsability of hardware design closer to application developers. Since it is a notoriously difficult and costly endeavour, their adoption has long been mostly limited to applications with extreme constraints and requirements.

[1]Such functionality has since been merged into general-purpose CPUs, but not, for example, in some Digital Signal Processors

The last fifteen years, however, have seen a renewed interest for custom accelerators. This may be explained by several factors. First, our computing needs have increased significantly, partly due to the growing amount of data produced each day, and the need to process them. Secondly, this period coincides with a turning-point in the hardware industry, with the end of the traditional scaling "laws" that have driven its development for more than 40 years.

In particular, the breakdown of Dennard scaling, which stated that power density (W/cm^2) would remain constant as transistor density increased, has had significant impact on both software and hardware design. Since *dynamic* (transistor-switching) power is proportional to clock frequency, manufacturers could exploit reductions in processor size to raise frequencies from one generation to the next without increasing the power budget. This resulted in regular performance upgrades for single-threaded code. Nowadays, however, *static* power is no longer negligible compared to dynamic power, mainly due to current leakages, and this strategy is no longer applicable.

Consequently, thermal dissipation is becoming a major issue. In fact, it is expected that, as transistor density continues to increase (albeit more slowly than before), a growing portion of integrated circuits will have to be turned-off at any given time to stay below nomimal thermal dissipation power (a phenomenon sometimes called "Dark Silicon"). Further improvements will then only come from better use of available transistors. Heterogeneous, accelerator-rich architectures are thus expected to become the norm. Unfortunately, designing hardware accelerators is still significantly more difficult than writing software. Lowering the barrier to entry, for example by developing new tools and methodologies, is thus an important challenge to address the needs of tomorrow's computing.

1.3 Accelerator Design

In both embedded systems and HPC, accelerator design may be stated as an optimization problem. One either seeks to minimize resource usage under performance constraints, or to maximize performance under resource constraints. In particular, when designing custom accelerators, the design space is extremely large, as many factors can influence the quality of the design. For example, wordlength may be reduced to trade accuracy for lower area cost, or local memory usage may be increased to tackle bandwidth limitations. The number of solutions is so large, one cannot expect to come up with an "optimal" or near-optimal design at first try; a time-consuming Design-Space Exploration (DSE) phase is usually required.

New methodologies are needed to enable and speed up this exploration. Traditional hardware design relies on the use of low-level Hardware Description Languages (HDLs), such as VHDL and Verilog, to specify the architecture. These Register-Transfer Level (RTL) languages provide a poor level of abstraction; in particular, cycle accuracy is part of their semantics, as state changes happen synchronously at clock signal edges. While this level of control is sometimes required, when designing accelerators, it typically hinders DSE by making it difficult to explore architectural variants.

In contrast, High-Level Synthesis (HLS) tools compile an algorithmic specifi-

cation (usually written in C or C++) to a low-level (RTL) description. With this approach, many implementation details are handled automatically by the tool, based on target frequency, hardware platform and designer directives. This rise in abstraction allows much faster DSE, as far-reaching, system-level architectural changes can be implemented in a few lines of code. Consequently, HLS can be combined with methodologies based on code generation to explore a large number of design points in a short amount of time.

1.4 Presentation of This Work

For efficient exploration, though, HLS alone is not sufficient. Tools are required to guide the designer and help him/her make the right implementation choices in each situation. As accelerators are used in different contexts (HPC vs. embedded systems), and since each application has unique characteristics (access patterns, arithmetic intensity, numerical stability), such tools must integrate domain-specific constraints to identify a suitable set of trade-offs between all performance metrics.

This thesis focuses on such DSE methodologies for *i*) fixed-point accelerators *ii*) FPGA accelerators for stencil computations. The document is organized as follows:

- The first two chapters are concerned with performance/accuracy trade-offs. Many applications can tolerate significant accuracy degradations before the quality of results is strongly affected. It is common to exploit this tolerance by converting floating-point applications to use fixed-point arithmetic, to benefit from its overall lower cost. Chapter 2 discusses this problem, and some methods to evaluate the accuracy degradation resulting from conversion to fixed-point. In Chapter 3 we present our contributions to the construction of analytical accuracy models for linear systems.
- The next chapters are concerned with implementation trade-offs for iterative stencil computations. Iterative stencil computations form a large class of algorithms with applications in scientific computing, embedded vision and more. They are presented in Chapter 4, along with implementation strategies. While many authors have proposed a "one size fits all" approach, in Chapter 5, we embrace the diversity of applications by proposing multiple architectural variants, along with associated performance models.
- We conclude in Chapter 6 with a review of our contributions and a discussion of potential perspectives.

Chapter 2

Accuracy Evaluation

2.1 Introduction

Floating-point arithmetic is based on a flexible approximation of real numbers offering sensible trade-offs between precision and representable range, freeing application developers from these concerns. Software programmers often forget about the complexity of the underlying machinery and take its almost universal support on general purpose hardware for granted. However, because of its significant resource cost, floating-point arithmetic is not always an option for embedded system designers. Instead, they must settle on less convenient, but more cost-effective fixed-point implementations.

Implementing fixed-point computations is inherently challenging, as the programmer or designer must take extra care to avoid numerical overflows while retaining enough accuracy for application requirements. At the same time, he/she must also ensure that design concerns, such as power consumption and area budget, are correctly addressed. Reconciling all these constraints at once is a difficult task, and applications are usually first specified, prototyped and functionally validated in floating-point arithmetic, with floating-point to fixed-point conversion handled at a later stage in the design flow.

Floating-point to fixed-point conversion exposes trades-offs between performance (area cost, power consumption) and accuracy. Design goals can be formalized as a constrained optimization problem. For example, one may wish to maximize accuracy under some fixed area budget, or minimize area cost subject to an application-specific accuracy constraint. The process of solving such problems is called *Word-Length Optimization* (WLO). Finding an optimal or near-optimal solution usually implies exploring a large design space, especially in a hardware design context where datapaths can be tailored to arbitrary bit-widths.

During WLO, the cost and accuracy of each candidate implementation must be assessed to determine whether it represents an improvement over the best known solution. Quick *accuracy evaluation* is especially challenging, as the impact of numerical errors on the output can be hard to predict. For this reason, bit-accurate fixed-point simulations are often used, but their poor performance combined with the large number of simulations required to produce reliable accuracy estimates leads to significant iteration times. As a consequence, WLO is often performed

semi-manually by expert designers, driving the exploration by identifying the most interesting design points. It is an error-prone, time-consuming task, taking up to 50% of overall design time [1], which is often interrupted as soon as a satisfying solution is found, leading to suboptimal implementations.

Combinatorial optimization algorithms can be used to perform WLO in a more systematic manner. In practice, because of long simulation times, this choice supposes the availability of a more efficient accuracy evaluation method. *Analytical* techniques try to solve this problem by constructing an *accuracy model* from a floating-point or infinite-precision specification, allowing the accuracy degradation associated to a fixed-point implementation to be estimated almost instantly, without simulations. Such methods have the potential to considerably improve the applicability of fully automated WLO. Unfortunately, as we will see in this chapter, they are currently limited to one-dimensional signal processing kernels and cannot properly handle higher-dimensional filters such as image or video processing algorithms.

2.2 Hardware Representation of Real Numbers

Implementing numerical computations implies choosing a finite, explicit approximation of real numbers. The two most popular options, fixed-point and floating-point arithmetic, have mostly opposite characteristics in terms of ease-of-use, hardware cost and numerical properties. After a brief review of fixed-point and floating-point arithmetic, this section describes their respective advantages and drawbacks, along with the trade-offs they expose.

2.2.1 Fixed-Point Arithmetic

In fixed-point arithmetic, real numbers are represented as integers, with an implicit scaling factor determining the position of the binary point. Concretely, let x be some arbitrary number and $I_{\hat{x}}$ its integral representation. The interpretation $\hat{x} \approx x$ is given by:

$$\hat{x} = \text{radix}^e \times I_{\hat{x}}$$

where radix is the base of the numeral system (usually 2) and e is a *fixed* exponent.

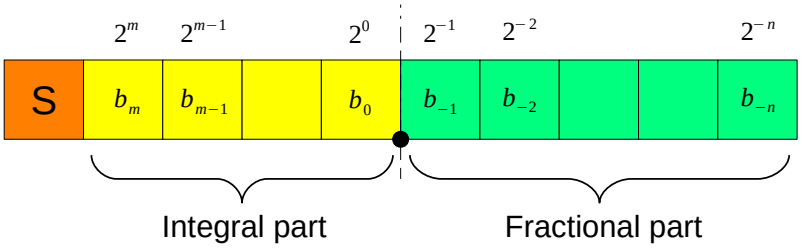


Figure 2.1: $Q_{m,n}$ format with m-bit integral part and n-bit fractional part.

Depending on implementations, $I_{\hat{x}}$ may be stored in two's complement representation, or as a sign-bit and an absolute value. When the binary point, determined by the scaling factor, falls in the middle of the representation, digits are partitioned into

integral and fractional parts, depending on their position. A fixed-point format with m-bit integral part and n-bit fractional part is often written $Q_{m,n}$ (see Figure 2.1).

Fixed-point arithmetic is implemented on top of integer arithmetic. Explicit rescaling operations must be performed to ensure compatibility of operands, control word-length or avoid overflows. For example, consider the (unsigned) fixed-point addition of $v_1 = 2^{-1} \times 10011b$ and $v_2 = 2^{-3} \times 01101b$. A custom word-length implementation is illustrated in Figure 2.2. The two numbers must first be aligned be to the same exponent by scaling v_2 down by two positions, which leads to the truncation of its two least significant bits. Finally, a sixth, *guard* bit is used to account for bit-growth and prevent overflows. This is not the only solution: for example, both operands could be further shifted by one position to keep output wordlength under 6 bits, or rounding could be used instead of truncation to improve error bounds.

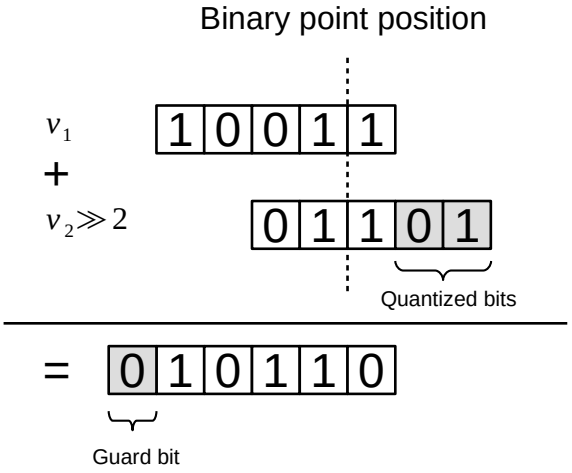


Figure 2.2: Fixed-point addition of two 5-bit numbers.

Fixed-point multiplication illustrates well the challenges of fixed-point arithmetic. Consider the product $v_1 \times v_2$, with v_1, v_2 defined as in the previous paragraph. No alignment is required to perform the operation, as:

$$v_1 \times v_2 = (2^{-1} \times 10011b) \times (2^{-3} \times 01101b) = 2^{-4} \times (10011b \times 01101b).$$

The result can thus be computed without loss of accuracy, irrelevant of the scaling factors. However, fixed-point multiplication of same word-length numbers produces a result of double bit-width, which can lead to a phenomenon sometimes called bit-width explosion. Additional truncations or roundings, called *quantizations*, must be introduced to avoid this problem. This leads to computational errors whose magnitude depends on the computation and the severity of the quantizations.

2.2.2 Floating-Point Arithmetic

Contrary to fixed-point arithmetic, where scaling factors are implicitly encoded in the computation, floating-point arithmetic uses explicit exponents in the representation itself in order to automatically scale to different ranges of values. It can be

Table 2.1: Interpretation of a IEEE 754 Floating-Point Number

Exponent Value	T	Interpretation	Remark
$e = emin$	0	$(-1)^S \times 0$	
$e = emin$	$\neq 0$	$(-1)^S \times 2^e \times 0.T$	Denormal numbers
$emin + 1 \leq e < emax$	any	$(-1)^S \times 2^e \times 1.T$	Normal numbers
$e = emax$	0	$(-1)^S \times \infty$	Infinities
$e = emax$	$\neq 0$	NaN	"Not a Number"

seen as a form of binary scientific notation. For example, whereas chemists refer to the Avogadro constant as:

$$6.02214086 \times 10^{23}\,\mathrm{mol}^{-1},$$

it can also be written in binary form as:

$$1.11111110000110000101111 \times 10^{1001110}\,\mathrm{mol}^{-1}.$$

More formally, a floating-point number consists in a sign bit S, a signed exponent E and a fixed-point number M with 1-bit integral part, called the *mantissa* or *significand*. The represented value is:

$$(-1)^S \times 2^E \times M$$

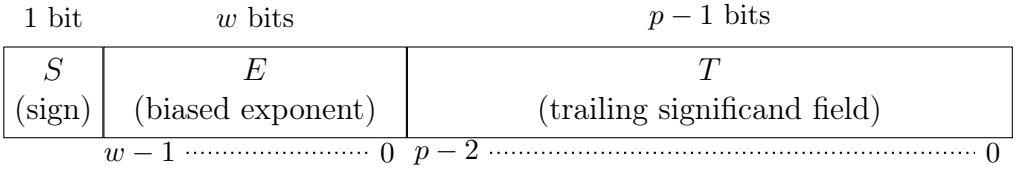


Figure 2.3: Binary Representation of a IEEE 754 Floating-Point Number

Most floating-point implementations are based on the IEEE754 standard and use the binary representation pictured in Figure 2.3. This encoding saves 1 bit by not storing the integral part of the significand, but inferring it from the fractional part and the exponent. The interpretation, detailed in Table 2.1, assumes that (most) floating-point numbers are stored in so-called normal form, which also ensures that they are represented with a maximum number of significand bits. The exponent e is stored as a biased integer e' such that $e = e' - 2^{w-1} + 1$. We write $e = emin$ when $e' = 0$, and $e = emax$ when $e' = 2^w - 1$

While floating-point arithmetic can be emulated on top of integer arithmetic, performance is prohibitive. Consequently, most implementations rely on dedicated hardware support, usually in the form of a Floating-point Processing Unit (FPU), off-the-shelf operators or as a custom datapath.

2.2.3 Comparison

In the following, we discuss the main differences between fixed-point and floating-point arithmetic with respect to programmability, hardware cost, availability and numerical properties.

Programmability

To the programmer or hardware designer, floating-point arithmetic offers many advantages in terms of simplicity, as floating-point hardware automatically performs necessary rescalings to maximize accuracy and minimize the risk of overflows.

In contrast, programming in fixed-point arithmetic often implies dealing with such problems manually. For example, multiplying two 8-bit unsigned fixed-point numbers, with respective exponents -2 and -4, may be written in C:

```
uchar8 mul_2_4(uchar8 a, uchar8 b) {
  return (a >> 4) * (b >> 6);
}
```

The programmer must manually keep track of the implicit factor of each datum in order to perform the right operation.

Libraries such as `SystemC` (OSCI), `ac_fixed` (Mentor Graphics) and `ap_fixed` (Xilinx) can handle some of these concerns for the hardware designer by performing automatic rescalings given the format of operands. However, fine-grained control of wordlength often requires the introduction of manual quantizations, expressed as intermediary variables which clutter the specification with implementation details.

Area Cost and Power Consumption

Floating-point hardware implementations are significantly more costly than fixed-point implementations. For example, floating-point adders require pre-alignment logic (usually in the form of expensive barrel shifters), adder/rounding logic and normalization logic with leading-zero detection. This makes floating-point arithmetic prohibitive for area-constrained applications. Power usage of floating-point operators is also typically higher than that of integer operators used in fixed-point arithmetic.

Availability

Because of their significant hardware cost, many embedded processors do not even feature FPUs. On these platforms, the use of fixed-point arithmetic is virtually mandatory.

Numerical Properties

In fixed-point arithmetic, the scaling factor is the weight of the least significant bit and corresponds to the smallest non-zero representable value, also called *quantization step*. Along with wordlength, it determines the range of representable numbers. For

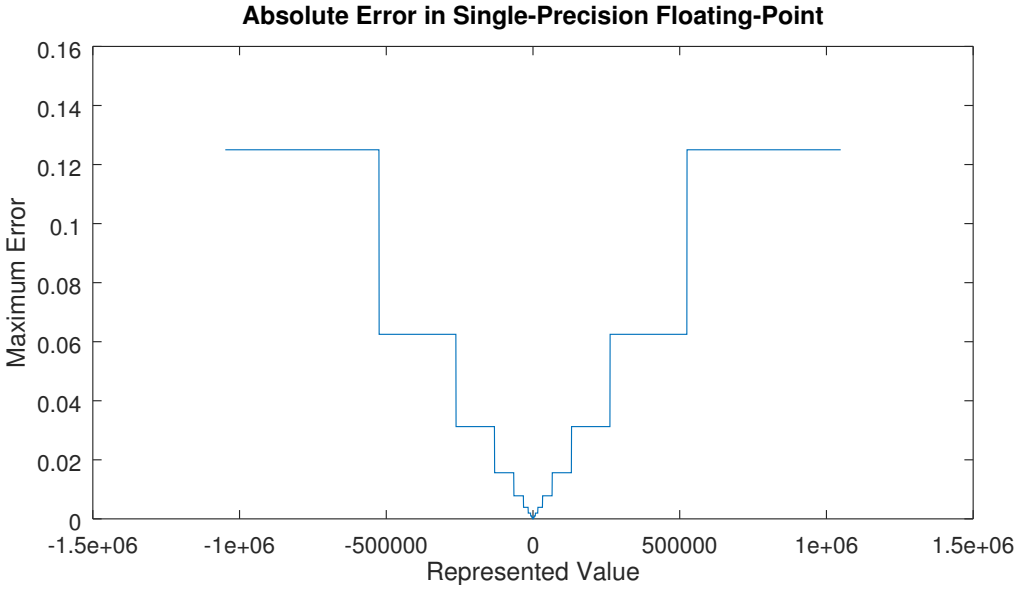


Figure 2.4: Maximum representation error of floating-point numbers as a function of represented value.

example, the $b = (m + n + 1)$-bit $Q_{m,n}$ format has quantization step 2^{-n} and covers the range:

$$[-2^{b-n-1}, 2^{b-n-1} - 1].$$

This interval can be extended by increasing the size b of the representation or choosing a coarser quantization step. This necessary trade-off between range and accuracy is a major disadvantage of fixed-point arithmetic.

In contrast, floating-point formats feature a *variable* exponent which allows them to represent a wide interval of numbers. Specifically, the range of (normalized) non-negative values that can be represented by a $b = (w + p)$-bit IEEE754 floating-point number is:

$$[2^{-(2^{w-1}-2)}, 2^{2^{w-1}} - 2^{2^{w-1}-p}]$$

Small values are encoded with a small exponent and considerable accuracy, while bigger exponents allow very large values to be represented, albeit possibly with larger errors. In other words, while fixed-point numbers have a fixed quantization step and bounded *absolute* errors, floating-point has quantization steps proportional to magnitude of numbers and bounded *relative* errors. Floating-point quantization step size as a function of number magnitude is plotted in Figure 2.4.

The notion of *dynamic range* can be introduced to summarize these properties. Dynamic range is defined as the ratio between the smallest and largest representable values and does hence not depend on the scaling factor of fixed-point formats. Dynamic range of fixed-point and floating-point formats are plotted in Figure 2.5 as a function of wordlength. We can observe that for very small bit-widths, fixed-point numbers actually offer a larger dynamic range. But as wordlength increases, this phenomenon is reversed and floating-point arithmetic gives much more flexibility.

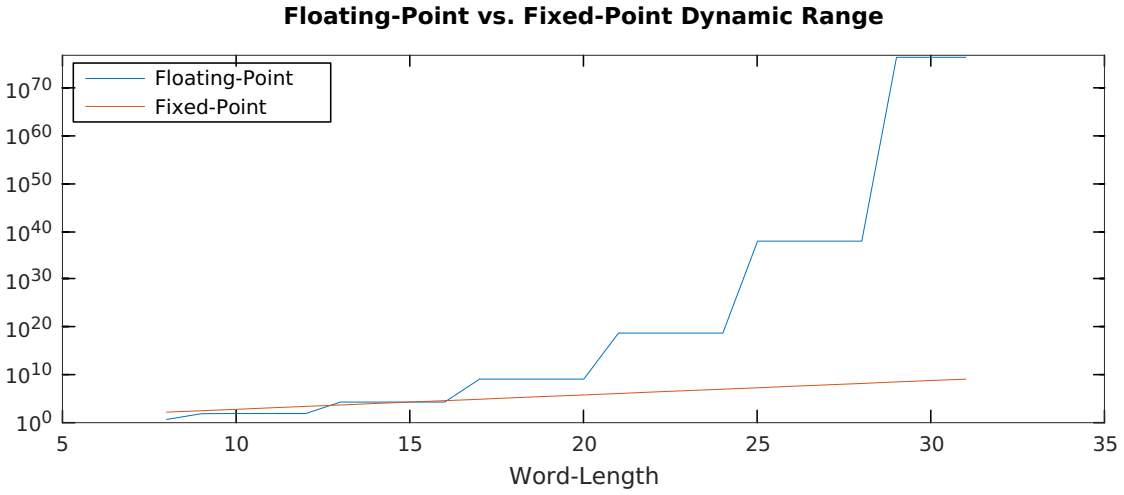


Figure 2.5: Dynamic range of floating-point and fixed-point representation as function of wordlength. Floating-point exponents of $\lceil 25\% \rceil$ of wordlength are assumed.

2.2.4 Summary

Floating-point arithmetic is more versatile than fixed-point arithmetic and offers better numerical properties, with a wide range of representable values and small relative errors. Unfortunately, its significant hardware cost hinders its adoption in many embedded contexts and fixed-point arithmetic must then be used.

2.3 Floating-Point to Fixed-Point Conversion

As many DSP processors lack support for floating-point arithmetic, or because of its significant area cost in FPGA and ASIC designs, many applications must be implemented in fixed-point. Unfortunately, programming in fixed-point arithmetic is a challenging task as overflows and numerical errors may significantly degrade accuracy. For this reason, these concerns are usually addressed at a later design stage: the application is specified and validated in floating-point arithmetic, before being converted to fixed-point.

In this section, we discuss the problem of floating-point to fixed-point conversions. We then expose the techniques available to address this problem automatically, with a focus on *accuracy evaluation*, a step that often is the bottleneck of the process.

2.3.1 Problem Setup

The intent of float-to-fix is to assign to each value in a computation a fixed-point format minimizing the risk of overflow and ensuring that enough accuracy is retained with respect to the reference implementation/specification. That specification may be given as a block diagram, Signal Flow Graph (see Section 2.4.1), or as a C/C++/Matlab source code.

Preventing overflows is usually achieved by performing *range analysis* on the input specification: the interval of each value (i.e., each signal in the graph, or each variable in the program) is determined in order to allocate enough bits in the most significant positions. Word-lengths are then adjusted until a suitable trade-off is found between performance/cost and accuracy. Depending on the context and target platform, the problem being solved can be stated differently:

- In a software/DSP setting, the challenge consists in finding a fixed-point specification that *i*) meets (or exceeds) accuracy requirements, *ii*) can be implemented using available CPU primitives.

  The design space is thus limited by the word-lengths and instructions proposed by the target processor.

- In a hardware design setting, floating-point to fixed-point conversion takes the form of a *Word-Length Optimization*, where one tries to minimize area cost *or* maximize accuracy, subject to some accuracy or cost constraint.

  The design space is usually considerably larger than in software fixed-point, as the datapath can be fine-tuned to arbitrary word-lengths. WLO problems are combinatorial in nature. It has been shown that a restricted analytical form of the multiple wordlength assignment problem is of NP-hard complexity [2].

Let $\mathbf{w}$ denote a fixed-point configuration, $\lambda(\mathbf{w})$ the associated error and $C(\mathbf{w})$ the cost estimate of that implementation. We distinguish two forms of WLO problems. The accuracy maximization problem may be stated as:

$$\max_{\mathbf{w}} \lambda(\mathbf{w}) \quad \text{subject to} \quad C(\mathbf{w}) \leq C_{max}$$

and the cost minimization problem as:

$$\min_{\mathbf{w}} C(\mathbf{w}) \quad \text{subject to} \quad \lambda(\mathbf{w}) \leq \lambda_{max}$$

These optimization problems can be solved using combinatorial optimization algorithms, or a more ad-hoc, semi-manual exploration. At a high-level, all approaches boil down to the same idea: starting from an initial configuration $\mathbf{w}_0$, the current solution is iteratively refined to optimize the objective function. At each step, the cost $C(\mathbf{w})$ and accuracy $\lambda(\mathbf{w})$ are evaluated and a new configuration is chosen until some acceptance condition is reached (for example, the absence of progress), or until the designer in charge of the conversion is satisfied with the results.

To perform this optimization automatically and reach a good solution in reasonable time, sufficiently fast methods are required for evaluating the cost and accuracy of a design. Cost estimation is a challenging task, as the actual cost may depend on decisions made by the design tool *after* float-to-fix conversion, such as operator-sharing. In practice, more-or-less comprehensive high-level cost estimates are used [3, 4].

While quantization noise can negatively impact the behavior of the system, *overflows* have an even stronger consequences on numerical correctness. Word-Length Optimization is mostly focused in tweaking the size of the fractional part of fixed-point operands to approach the optimal solution. However, before entering the

optimization loop, it is necessary to determine the size of the integral part, in order, depending on the criticity of the application, to limit or the risk or guarantee the absence of overflows. This analysis is called *range estimation*.

2.3.2 Range Estimation

In order to avoid overflows, it is necessary to ensure that the range of values spanned by each variable during the computation does not exceed the capacity of its representation. This range naturally depends on inputs and can be estimated from their own range or from representative samples.

When input range is known, any static analysis designed to compute safe variable bounds can be used. *Interval* or *Affine Arithmetic* have been extensively applied to this problem. Affine arithmetic can model exactly range propagation through a linear non-recursive program, but non-linearity or the presence of feedback loops gives rise to approximations.

For LTI systems, the L^1 norm of the transfer function (which can be computed by hand, or automatically from an adequate representation) gives precise information on the range of outputs. David Cachera and Tanguy Risset proposed a formal approach based on the polyhedral model and the $(\max, +)$ tropical algebra to compute ranges on affine loop nests operating on uni-dimensional arrays [5].

In general, without stringent restrictions on the nature of the system, any safe static method is bound to produce pessimistic over-approximations: computing the precise semantics of a program is an undecidable problem. Moreover, even when error bounds *can* be determined exactly (for example, using affine arithmetic in a basic block with linear operations), numbers close to the minimum or maximum values are unlikely to be observed in practice, as they correspond to statistical extremes.

As a constrained example, consider the addition of 5 independent uniform random variables ranging over interval $[0, 1]$. Their sum is obviously distributed over interval $[0, 5]$. However, as evident from the plot of its probability density function (see Figure 2.6), values over 4 are unlikely to be observed - in fact, the probability is less than 1%. One may choose to use saturating arithmetic and only assume values less than 4, without significantly affecting the results of the computation. However, purely static methods are unlikely to help the designer to recognize such situations.

Except in critical systems, where overflows are not acceptable, simulation is thus often preferred, or used in complement, to static analyses: provided that inputs are in a sufficient number and statistically representative, measured bounds indirectly reflect signal characteristics, and are thus often much more tight than those obtained with static approaches.

2.3.3 Accuracy Metrics

Formulating an accuracy constraint supposes the choice of a particular metric to characterize performance degradations. Two main classes of metrics may be used:

"Hard" metrics (error bounds) In critical systems, accuracy constraints are usu-

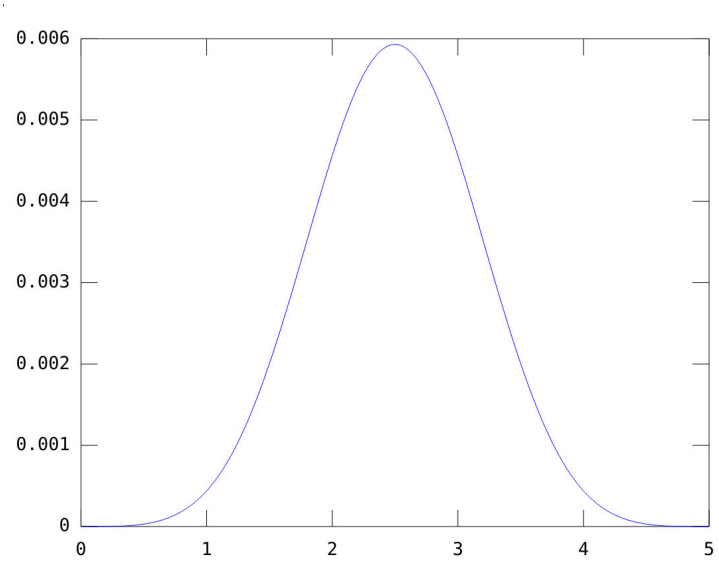


Figure 2.6: PDF of the sum of 5 i.i.d random variables with distribution $\mathcal{U}([0,1])$.

ally specified as a hard bounds on error. Typically:

$$|e| = |\hat{x} - x| < \varepsilon$$

Statistical metrics In signal and image processing systems, soft metrics based on the statistics of signals are usually used. The most common one is called *noise power* and involves the first and second statistical moments of the error, seen as a random noise e:

$$P(e) = E(e^2) = \mu_e^2 + \sigma_e^2$$

where μ_e and σ_e^2 denote the mean and variance of variable e. Noise power is generally given in decibels (dB):

$$P_{\log}(e) = 10 \log_{10} P(e) \text{ dB}.$$

A related way to measure the relative magnitude of signals and errors is the *Signal to Quantization Noise Ratio* (SQNR). It is defined as:

$$\text{SQNR} = P_{\log}\left(\frac{\text{Signal}}{\text{Error}}\right) = P_{\log}(\text{Signal}) - P_{\log}(\text{Error})$$

Signal power $P_{\log}(\text{Signal})$ is generally known from representative inputs. Computing noise power or SQNR is thus equivalent.

Finally, whereas in noise power, only the first two moments are used, estimates of higher-order moments give more information on the shape of the probability distribution and can be used to define even more fine-grained constraints [6]. However, in the following, we will mostly consider noise power, as noise power and SQNR are the most widely used metrics.

2.3.4 Accuracy Evaluation

Accuracy evaluation is the process of evaluating the accuracy degradation occasioned by a fixed-point implementation.

Simulation

Given a fixed-point specification $\mathbf{w}$, the obvious way to determine its accuracy is to perform bit-accurate fixed-point simulations with representative inputs and compare the results with the reference implementation. This approach produces reliable estimates and is easy to implement for any computation. However, it is also very time-consuming:

- Compared to floating-point or native integer operations, fixed-point simulations suffer from a large performance hit on general purpose hardware.
- To produce reliable estimates of statistical metrics, this process must be repeated a large number of times to determine the statistical moments of the error with enough confidence.

Since accuracy evaluation is performed at every WLO step, the use of simulations is often a bottleneck limiting the depth of the design space exploration, leading to suboptimal implementations

Analytical Approaches

In analytical methods, an *accuracy model* of the specification is constructed prior to WLO to avoid simulations and speed up accuracy evaluation. While the construction of the model may be relatively costly, it can be used to quickly determine the accuracy of any solution, thus considerably increasing the number of optimization steps that can be performed.

2.4 Analytical Accuracy Evaluation

Our contributions, exposed in the next chapter, focus on analytical accuracy evaluation. The principle of analytical accuracy evaluation is to derive an *accuracy model* from a floating-point specification. The statistical moments of quantization errors, viewed as random variables, are propagated through the computation to construct a symbolic expression of overall noise power at the output of the system.

Two kinds of model are required: first, the statistical properties of quantization errors need to be determined. Secondly, the overall impact of the system on these errors at the output must be captured by abstract models.

Current methods operate on dataflow models such as Signal Flow Graphs as an intermediate representation of the system (Section 2.4.1). These graphs are transformed with simple rewrite rules to introduce *error sources* (Section 2.4.2) representing quantization errors as additional inputs. Quantization noise models (Section 2.4.3) provide expressions for the mean and variance of these errors as a function of input and output precisions. The challenge then consists in constructing a noise

formula representing the moments of errors at the output of the system. A variety techniques have been proposed to achieve this goal, with different assumptions on the system. They are discussed in the rest of this section.

2.4.1 Signal-Flow Graphs

Signal Flow Graphs [7] (SFGs) are a flavor of synchronous data flow [8] graphs used in the signal processing community to model discrete computations. Semantically, each node in a SFG represents a sequence of values, defined in terms of the node's predecessors. In particular, SFGs contain explicit delay operations, in the form of "register" nodes (usually marked z^{-1}): at any point in time, the output of these nodes is defined as their input at the previous clock cycle.

As an example, an SFG for a Finite Impulse Response (FIR) filter is shown in Figure 2.7. The input sequence $x(n)$ is delayed through a series of register nodes which can collectively be seen as a shift register. The output $y(n)$ is defined as the dot product of the content of the shift register and a vector of coefficients $(b_i)_{0 \leq i \leq 3}$.

Alternatively, SFG nodes may be expressed as recurrence equations. For example, the SFG in Figure 2.7 is a graphical representation of the following system:

$$\begin{cases} y(n) = m_3(n) + p_2(n) & m_2(n) = b_2\delta_2(n) \\ p_2(n) = m_2(n) + p_1(n) & m_3(n) = b_3\delta_3(n) \\ p_1(n) = m_1(n) + m_0(n) & \delta_3(n) = \delta_2(n-1) \\ m_0(n) = b_0x(n) & \delta_2(n) = \delta_1(n-1) \\ m_1(n) = b_1\delta_1(n) & \delta_1(n) = x(n-1) \end{cases}$$

SFGs are schedulable if any cycle contains at least one delay node, while graphs with 0-weight cycles do not represent any meaningful system. An SFG verifing this condition is unambiguously defined modulo initial conditions – the state of the system before the beginning of the computation. This validity condition may be seen as a restriction of the conditions [9] given by Karp, Miller and Winograd for a system of uniform recurrence equations to be explicitly defined.

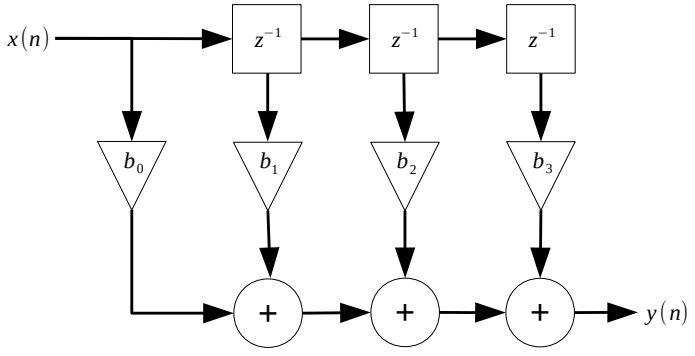


Figure 2.7: SFG of a FIR Filter computing the formula: $y(n) = \sum_{i=0}^{3} b_i x(n-i)$. Nodes labeled z^{-1} represent one-cycle delays and triangle-shaped nodes multiplication by a constant coefficient.

```
#define N 4

#pragma MAIN_FUNC
float fir8() {
#pragma DYNAMIC [-1, 1]
    float sample;

#pragma OUTPUT
  float acc;
  int i;

#pragma DELAY
  static float X[N];
  X[0] = sample;

  acc = X[N - 1] * b[N - 1];

  for (i = N - 2; i >= 0; i--) {
    acc += X[i] * b[i];
    X[i + 1] = X[i];
  }
  return acc;
}
```

Figure 2.8: FIR filter implementation for the Id.Fix conversion tool.

Some tools such as Id.Fix [10] build a SFG out of a annotated C program. After WLO, a C/C++ fixed-point implementation, using the `ac_fixed` library, is output. There are two main advantages to this approach. First, a source code implementation is more easily integrated into a custom validation framework than a SFG or a block diagram. Perhaps more importantly, this technique can be used in a HLS context, with WLO viewed as a *source-to-source* transformation. In Figure 2.8, an implementation of the FIR filter in Figure 2.7 is given, as accepted by Id.Fix:

This code actually represents one iteration of the FIR. After parsing, the control flow of the top function (marked by the `MAIN_FUNC` pragma) is fully flattened and an acyclic data flow graph is built with a producer-tracking simulation. Finally, the `DELAY` pragma helps the tool insert the delay nodes, corresponding to dependences across consecutive function calls.

2.4.2 Error Sources

In a fixed-point implementation, an arithmetic operator can introduce multiple errors: inputs may need to be quantized to fit the operator's format and the precision of the output may also be reduced to limit bit-width growth.

Quantizations may be expressed explicitly as additional operations, as shown in Figure 2.9. In analytical accuracy evaluation, though, round-off errors are modeled as *additive noise* perturbating an infinite-precision signal. This is usually reflected through a graph transformation: each quantization is replaced with an addition

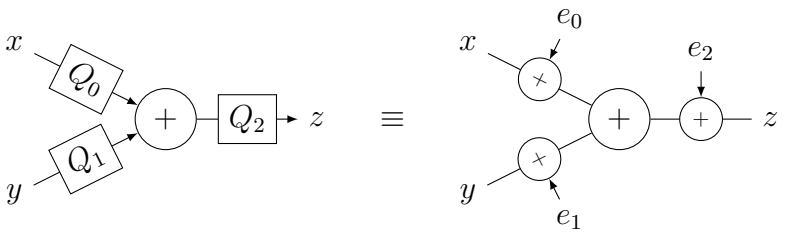


Figure 2.9: Introduction of error sources in a Signal Flow Graph.

between the original signal and the quantization error. The virtue of this transformation is to reframe quantization errors as new system inputs, which can be modeled as a stochastic process known as *Pseudo Quantization Noise* (PQN).

2.4.3 Pseudo Quantization Noise model

Analytical accuracy evaluation seeks to predict the influence of quantization errors on the output of the system. At first, this may appear like an infeasible task, since actual errors depend on system inputs. As it turns out, in the vast majority of cases, quantization errors can be statistically characterized from the precision of the original and quantized signals, and the mode of quantization (truncation, rounding or convergent rounding). Moreover, this *Pseudo Quantization Noise* is uncorrelated from the input signal and other error sources, which further simplifies the analysis.

For example, consider the truncation of some infinite-precision signal x to x', with quantization step $q = 2^{-n}$. We have:

$$x' = x + e_x$$

with error $e_x = x' - x$ within the interval:

$$I =]-q;\ 0[.$$

It can be shown [11] that, if quantization step q is sufficiently small, e_x can be modeled as uniformly distributed variable:

$$e_x \sim \mathcal{U}(I)$$

such that signal x and quantization noise e_x are uncorrelated. We can thus give the mean and variance of the error:

$$E(e_x) = -q/2 \qquad \mathrm{Var}(e_x) = \frac{q^2}{12}$$

The model above captures the distribution of errors as a continuous probability distribution, and is thus suitable for modeling the quantization of an infinite-precision (analog, floating-point) signal to fixed-point precision. Using rounding instead of truncation leads to a similar model with $I = (-q/2; q/2]$ and $E(e_x) = 0$, whereas discrete distributions can be used to characterize round-off errors between fixed-point formats [12]. Noise models for different configurations are shown in Figure 2.10.

Quantization Mode	Mean		Variance	
	discrete	continuous	discrete	continuous
Truncation	$-\frac{q}{2}(1-2^{-k})$	$-\frac{q}{2}$	$\frac{q^2}{12}(1-2^{-2k})$	$\frac{q^2}{12}$
Rounding	$-\frac{q}{2}\times 2^{-k}$	0	$\frac{q^2}{12}(1-2^{-2k})$	$\frac{q^2}{12}$
Convergent Rounding	0	0	$\frac{q^2}{12}(1-2^{-2k})$	$\frac{q^2}{12}$

Figure 2.10: PQN characteristics based on input / output signal precision and quantization mode. q represents the quantization step and k the number of eliminated bits when converting between fixed-point formats. Note that when $k \to \infty$, the discrete model converges towards the continuous model.

2.4.4 Operator-Level Noise Propagation

Under certain conditions, noise mean and variance can be naturally propagated through linear operations (addition of signals and multiplication by a constant). In particular, if $x' = x + e_x$, $y' = y + e_x$, then:

$$\lambda x' + y' = (\lambda x + y) + e_{\lambda x+y}$$

where $e_{\lambda x+y} = \lambda e_x + e_y$. Thanks to the linearity of the expected value operator,

$$E(e_{\lambda x+y}) = E(\lambda e_x + e_y) = \lambda E(e_x) + E(e_y)$$

and the mean of the error at the output can thus be computed from the average error of the input signals.

Variance propagation is a bit more subtle. If X and Y represent two *uncorrelated* random variables, then:

$$\mathrm{Var}(\lambda X + Y) = \lambda^2 \mathrm{Var}(X) + \mathrm{Var}(Y).$$

When input errors are known to be independently distributed, this formula may be used to derive the output error variance of linear operators. Unfortunately, this is not true when noises are, in fact, correlated. This may happen even when all noise sources are independently distributed. For example, consider the degenerate case where $\lambda = 1$ and $X = Y$. X and Y denote the same random variable and are thus obviously correlated. We have:

$$\mathrm{Var}(X + Y) = \mathrm{Var}(2 \times X) = 4 \times \mathrm{Var}(X) \neq \mathrm{Var}(X) + \mathrm{Var}(Y) = 2 \times \mathrm{Var}(X)$$

which only holds if $\mathrm{Var}(X) = 0$.

Operator-level propagation of noise statistical parameters is used by Tourreilles et al. [13] to implement a Word-Length Optimization pass within the GAUT HLS framework. However, the assumption is made that errors are always independent. In some cases, this can lead to large over- or under-approximations of noise power.

The following conditions together guarantee the absence of correlations between errors within a SFG, and can thus be used to verify whether the simple noise propagation method exposed above is applicable.

- All noise sources are uncorrelated.

- Distinct paths between the same two nodes contain different numbers of delays.

The second condition ensures that a single value cannot contribute twice to the same error. It is verified, for example, by the FIR filter in Figure 2.7. However, many computations do not possess this property ; in such cases, accuracy estimation must capture correlations between noises to produce reliable results. This is a global property of the graph and cannot be handled at the operator level.

Another major difficulty is the presence, in many applications, of non-linear operations such as multiplications between signals. Indeed, we have:

$$(x + e_x) \times (y + e_y) \approx xy + (xe_y + ye_x), \tag{2.1}$$

(where the term $e_x e_y$ is deemed negligible and voluntarily omitted). The error term $xe_y + ye_x$ depends not only on error signals e_x and e_y, but also on signals x and y. To address this issue, many techniques are restricted to linear systems, while others try to fallback to the linear case through linearization.

2.4.5 Noise Propagation in Linear Systems

Linear systems form a convenient framework for the study of error propagation. In such algorithms, signals and errors do not interfere and may be considered independently. Let T be linear system, $\mathbf{x}$ an input signal and $\widehat{\mathbf{x}} = \mathbf{x} + \epsilon$ the same input perturbated by some random noise ϵ. By definition of linearity:

$$T(\widehat{\mathbf{x}}) - T(\mathbf{x}) = T(\epsilon)$$

In other words, the propagated error is simply the output of the system when applied to the input error.

A linear system is called *Linear, Time-Invariant* (LTI) if a translation of its input by a constant offset results in the same offset at the output. Formally:

$$\forall \delta, \quad T(\tau_\delta(\mathbf{x})) = \tau_\delta(T(\mathbf{x})),$$

where τ_δ represents the translation of a signal by δ:

$$\tau_\delta(\mathbf{x})(n) = \mathbf{x}(n + \delta).$$

The temporal behavior of an LTI system $\mathbf{y} = T(\mathbf{x})$ is fully characterized by its *impulse response* $\mathbf{h}$, i.e., the output of the system when stimulated by a *unit impulse*. For any input $\mathbf{x}$:

$$\mathbf{y} = \mathbf{h} * \mathbf{x},$$

where $*$ denotes the convolution operation. Equivalently, the transfer function $\mathbf{H}$, defined as the Z-transform of the impulse response $\mathbf{h}$, contains the same information in the *complex-frequency* domain, where multiplication replaces convolution:

$$\mathbf{Y} = \mathbf{HX}$$

.

A well-formed single-input, single-output (SISO) SFG where the only operations (besides delays) are additions and multiplications by scalar values always represents an LTI system[1]. We give an intuition of the proof, by inductive reasoning on the structure of the graph, in the case of a non-recursive system:

- If the output node is also the input node, then the computation is the identity transformation $x(t) \mapsto x(t)$ which is obviously LTI.
- Otherwise, suppose the hypothesis true for each sub-SFG induced by all the ancestors of one of the output's predecessors (i.e., the sub-graphs computing the operands of the output node). Proceeding by case analysis:
  - If the output is a delay, let T be the system represented by the subgraph corresponding to its unique predecessor. Then, the SFG implements the system: $\tau_{-1} \circ T$.
  - Similarly, if the output is a linear operation $(x, y) \mapsto \lambda x + y$, let T_x and T_y be the systems corresponding to its operands. The implemented system is: $\lambda T_x + T_y$.

For example, the FIR filter represented by the SFG in Figure 2.7 is an LTI system.

In general, a multiple-input, multiple-output (MIMO) SFG verifying the same conditions as above (well-formedness, linear operators) does not represent an LTI system *per se*, but can be modeled as a combination of LTI systems. More precisely, let $T : (x_1, \ldots, x_m) \mapsto (y_1, \ldots, y_n)$ be the mapping between signals represented by the SFG. For any $i \in \{1, \ldots, n\}$, there exists m LTI systems $T_{i,1}, \ldots, T_{i,m}$ such that:

$$y_i = \sum_{j=1}^{m} T_{i,j}(x_j).$$

In other words, each input contributes *additively* to each output. In the temporal and frequency domain, we may also write:

$$y_i = \sum_{j=1}^{m} h_{i,j} * x_j \qquad Y_i = \sum_{j=1}^{m} H_{i,j} X_j.$$

This observation is key to analyze error propagation in an LTI system. Indeed, after modeling quantization errors as additional inputs (see Section 2.4.3), the SFG can be seen as a MIMO system. Since PQN-modeling only introduces additions, the propagation of each error to each output can be modeled as an LTI-system, and thus be fully captured by the corresponding impulse response or transfer function.

This approach is used in [14] to implement automatic wordlength optimization on an annotated Simulink block-diagram using transfer functions. However, the computation of these transfer functions is not detailed, and may not be automatic. Menard et al. [15] proposed a similar approach base on graph algorithm computing the transfer functions. The SFG is decomposed into acyclic subgraphs whose transfer functions are recursively computed and combined into a single one modeling the propagation of each noise source.

[1] The converse is not true.

2.4.6 Noise Propagation in Non-Linear Systems

As mentioned in Section 2.4.4, non-linear operations such as multiplication between signals introduce a problematic dependency between signals and noise propagation.

Constantinides et al. [16] proposed to recast non-linear systems as linear *time-varying* systems to apply some results on LTI systems to non-linear algorithms. Their approach supposes that each node represents a differentiable operation:

$$y(t) = f(x_1(t), \ldots, x_n(t)).$$

Since error values $\epsilon_1(t), \ldots, \epsilon_n(t)$ are small in comparison with signals, a first-order Taylor approximation of the output error is given by:

$$\epsilon(t) \approx \frac{\partial f(t)}{\partial x_1}\epsilon_1(t) + \cdots + \frac{\partial f(t)}{\partial x_n}\epsilon_n(t)$$

For example, if $f(x_1, x_2) = x_1 \times x_2$:

$$\epsilon(t) \approx x_2(t)\epsilon_1(t) + x_1(t)\epsilon_2(t)$$

(Note that this expression is essentially the same as Equation 2.1.)

This *small-signal* model is a linear function of input errors with time-varying coefficients: as evidenced by the above example, the partial derivatives depend on the value of t. To account for this fact, values of the derivatives are computed numerically through simulation with sufficiently large representative inputs. The SFG is then transformed into its corresponding small-signal model, with derivatives as input coefficients. For each noise source, another simulation is run with a noise of known mean and variance as input. The statistical moments of the output are then computed, and linearity is used to *i*) scale the results to noises of different mean and variance *ii*) build an accuracy model reflecting the *additive* contribution of each noise source as a function of fixed-point encodings.

This method can be seen as a hybrid simulation-based and analytical method: whereas simulations are used to construct the accuracy model, none are required during accuracy evaluation. Unfortunately, they make the unrealistic assumption that variance contributions of different noise sources can be summed, implicitly supposing that they are uncorrelated. There is no validation of accuracy estimates against simulations. Their propagation model is thus similar to that of Tourreilles [13] and of probably limited applicability.

In other approaches [17,18], a system is characterized by its time-varying impulse response. Expressions of noise power are then derived from signal statistics (cross-correlation, second-order moments) computed after a single floating-point simulation. Finally, an approach based on Affine Arithmetic [19] has been proposed. We believe this approach to be fundamentally tied to that of Rocher et al. [18], with correlation between signals captured by a different formalism.

2.4.7 System-Level Approaches

To handle large systems made of several sub-components and manage the combinatorial explosion of the design space, a hierarchical, *divide-and-conquer* approach

may be beneficial [20]. The WLO process is decomposed into sub-problems where each component is assigned an accuracy budget. The issue is that the output noise of sub-systems is not uniformly distributed. Its statistical distribution must be captured by different means, for example with PDF-shaping [6] or by its spectral power density [21].

2.5 Limitations of Analytical Accuracy Evaluation

The analytical accuracy evaluation techniques overviewed in this chapter are intrinsically limited by the use of SFGs as an intermediate representation. Indeed, this representation can only *compactly* model one-dimensional systems is thus not suitable for image and video processing applications. While SFGs can in fact be built from such algorithms by unrolling the full computation, their size is proportional to the dimensions of the image/scene – while the SFG of an IIR filter does not depend on the length of the sequence – which leads to severe scalability issues when constructing the accuracy model.

In trivial cases, such as the convolution of an image by a mask of coefficients, this problem can be sidestepped by restricting the analysis to the computation of a single element. For example, the kernel of a Gaussian blur filter may be extracted into the following C function:

```
pixel_t gaussian_blur(pixel_t pxs[9]) {
  return ( 0.025*pxs[0] + 0.108*pxs[1] + 0.025*pxs[2]
         + 0.108*pxs[3] + 0.469*pxs[4] + 0.108*pxs[5]
         + 0.025*pxs[6] + 0.108*pxs[7] + 0.025*pxs[8] );
}
```

which is seen by tools such as ID.Fix [10] as a multiple-inputs, single-output LTI system. Assuming that input quantization noise is spatially uncorrelated and identically distributed, as per the Widrow hypothesis, output noise does not depend on the position in the image and the accuracy model built for this function may be used to estimate the accuracy of the full algorithm. However, this strategy suffers from several shortcomings.

The first one is that *output* noise is almost always spatially correlated, even when input noise is not. For example, consider two adjacent pixels in the output of the Gaussian filter: since they are computed from overlapping windows of inputs, a large error in one value can strongly affect *both* results, and the errors are thus correlated. It is not a problem when processing a single filter, as noise power or SQNR can still be obtained from the statistical moments determined by the accuracy model. However, it means that the trick above can not be repeatedly applied to handle image processing pipelines made of the composition of multiple filters.

Another issue is that more complex cases, such as recursive algorithms, cannot be conveniently expressed as a simple function as above. Unfortunately, the scalability issues of current methods are even more important for these cases, as the number of propagation paths to be considered grows exponentially with the size of the image.

Finally, the impact of the quantization of constant coefficients is not fully explored by previous approaches. In methods handling LTI systems [14, 15, 22, 23],

it is usually assumed that the sensitivity of the transfer function to the quantization of coefficients has been assessed before floating-point to fixed-point conversion. Accuracy evaluation tools should also help the designer in this process.

Solutions to some of these problems are discussed in the next chapter.

Chapter 3

Improving Applicability of Source-Level Accuracy Evaluation

3.1 Introduction

In the previous chapter, we exposed the Word-Length Optimization and accuracy evaluation problems. We described a range of *analytical* techniques to derive closed-form accuracy models from floating-point specifications. We saw that such models can considerably speed-up design space exploration and allow for better implementations – however, their elaboration requires a precise modeling of the computation, a technical challenge that currently limits the scope of analytical methods to small sets of problems.

Recent work [REFs] has focused on statistical modeling of decision (branching) operators to handle non-static, data-dependent control flow. To our knowledge, noise propagation through arithmetic operations with singularities, such as division, is still difficult to capture reliably. In this chapter, we extend the applicability of analytical approaches in another direction. Specifically, we add support for multi-dimensional algorithms, such as image or video processing filters.

As a first step in this endeavor, we focus on *Linear Shift-Invariant* (LSI) filters, a multi-dimensional generalization of LTI systems. This class of computations is exemplified by the Deriche edge detector, a recursive image filter beyond the reach of previous approaches. This algorithm is discusssed in Section 3.2. In Section 3.3, we proceed with a more formal account on LSI systems. Our contributions *per se* are described in Section 3.4. They can be summarized as follows:

- We replace SFGs with *Multi-Dimensional Flow-Graphs* (MDFGs) as an intermediate representation of the system/program. This representation allows us to capture regular access patterns over multiple dimensions and is thus well-suited to model the class of computations we target.
- We propose a recursive algorithm to efficiently compute all the multi-dimensional transfer functions within a MDFG. These transfer functions compactly model the propagation of quantization noise between any two operations.
- We (partially) address the problem of *inferring* MDFGs from source code specifications. Specifically, we borrow a powerful dependence analysis technique

from the polyhedral compilation toolset to transform *affine* loop-nests into systems of recurrence equations. The MDFG is retrieved through a series of equational, semantic-preserving transformations.

- Finally, we present a methodology to handle the quantization of coefficients prior to Word-Length Optimization, by automatically computing the frequency response of the modified system.

We present experimental results in Section 3.5. We discuss future work, extensions and some open problems in Section 3.6 and conclude in Section 3.7.

3.2 Motivating Example: Deriche Filter

The Deriche or Canny-Deriche edge detector is a recursive image filter that cannot be handled by current techniques. It constitutes a good example of a Linear Shift-Invariant algorithm. Like most edge detection techniques, the Deriche filter proceeds by computing the gradient field of the image. Horizontal and vertical gradients G_x and G_y are compted independently. By symmetry, computing G_y is the same as computing G_x on the transpose of the image: from now on, we thus focus on horizontal gradient G_x to simplify the discussion.

The algorithm can be decomposed in two groups of recursive passes. The image is first processed in both horizontal directions (left-to-right and right-to-left). The results are then summed and the output is processed similarly along the vertical axis[1]. More precisely, the computation can be described by the following equations, where I represents the input image:

$$
\begin{aligned}
x_1(i,j) &= a_1 I(i,j) + a_2 I(i-1,j) + b_1 x_1(i-1,j) + b_2 x_1(i-2,j) \\
x_2(i,j) &= a_3 I(i+1,j) + a_4 I(i+2,j) + b_1 x_2(i+1,j) + b_2 x_2(i+2,j) \\
x &= x_1 + x_2 \\
y_1(i,j) &= a_5 x(i,j) + a_6 x(i,j-1) + b_1 y_1(i,j-1) + b_2 y_1(i,j-2) \\
y_2(i,j) &= a_7 x(i,j+1) + a_8 x(i,j+2) + b1 y_2(i,j+1) + b_2 y_2(i,j+2) \\
G_x &= y_1 + y_2
\end{aligned}
$$

These equations describe the flow of data along with the actual operations. Some of them are recursively defined (x_1, x_2, y_1 and y_2) ; for this specification to be computable, their value must hence be explicitly given outside some region. We simply define $\_(i,j) = 0$ whenever $(i,j) \notin [0; W-1] \times [0; H-1]$, where W and H are symbolic constants representing the dimensions of the image.

The expression of the coefficients $a_1, \ldots, a_8$ and b_1, b_2 depends on a scalar parameter $\alpha > 0$, defining the amount of smoothing applied prior to gradient computation. With other edge detectors such as the Sobel filter, a smoothing pass is usually needed as a pre-processing phase to remove high-frequency noise that can could cause false edge detections. In the Deriche filter, low-pass filtering and gradient computation

[1]Because of separability properties of the filter, the order between vertical and horizontal passes is mostly arbitrary.

are combined into a single step. The coefficients of the recurrence equations are defined such that the frequency response of the filter reflects the combination of the two phases. The main benefit is that the number of operations per pixel is not affected by the amount of smoothing required. In a non-recursive implementation with a convolution kernel, the size of the mask can vary greatly depending on the value of α. This makes the Deriche filter a great fit for noisy images, requiring a large amount of filtering. Finally, this algorithm exhibits interesting signal-processing properties: as the smoothing (i.e., noise filtering) filter is recursively implemented, quantization noise itself tends to be filtered out in later computations.

These qualities make the Deriche filter an interesting choice for limited-wordlength implementations on DSP processors or FPGAs, whose design phase often includes a floating-point to fixed-point conversion step. Unfortunately, earlier analytical accuracy evaluation methods are unable to construct an accuracy model for the Deriche filter or similar algorithms. Indeed, SFGs can only represent regular computations such as Deriche by fully flattening the control flow, thus distinguishing the computation of individual pixels and intermediary values. This results in a very large graph, where the number of propagation paths increases exponentially with each dimension of the image because of recursivity. Current techniques are thus affected by severe scalability issues when processing multi-dimensional algorithms.

Besides multi-dimensionality, Deriche exhibits another feature that cannot be appropriately captured by SFGs: *non-causality*. Fixing some column index j, let us write $x_2(i,j) = x_{2,j}(i)$. The equation $x_{2,j}$ defines an LTI system, specifically an *Infinite Impulse Response* (IIR) filter. However, interpreting i as the time time dimension, the processing order imposed by the recurrence is reversed: $x_{2,j}(i)$ depends on $x_{2,j}(i+1)$. This may be viewed as a dependence on "future" outputs, and could be modeled as a SFG by reversing the interpretation of delay nodes. Now, consider the summation of $x_{1,j}$ and $x_{2,j}$ in $x = x_1 + x_2$: as the summation of two LTI filters, it is also LTI, but each output now depends simultaneously on the "past" **and** the "future". Signal Flow Graphs are designed to model streaming computations, and cannot handle such dependence patterns. In other words, some one-dimensional LTI filters cannot be compactly represented as SFGs either.

In the following, we address these difficulties by using a more suitable representation that doesn't require any unrolling/flattening and does not impose a single, synchronous execution order. The class of systems currently supported by our approach, called Linear Shift-Invariant systems, is described in the next section.

3.3 Linear Shift-Invariant Systems

Some basic notions on LTI systems have already been defined in the previous chapter. However, in order to make the description of our contributions clearer, we want to take the time to discuss LTI and LSI systems in a more formal way.

3.3.1 Signals

An n-dimensional *signal* (or nD-signal for short) is a mapping from n-dimensional coordinates to scalar values. For example, a time-varying signal is a 1D-signal

mapping time to values, whereas a gray image is a 2D-signal from pixel coordinates to intensity levels.

Signals can be defined on continous ($\mathbf{R}^n$) or discrete ($\mathbf{Z}^n$) domains. In computers, though, we almost always process discrete signals. In the following, an nD-signal thus denotes a function: $\mathbf{Z}^n \to \mathbf{R}$. If a signal x is only defined on some subset $D \subset \mathbf{Z}^n$, we extend it to $\mathbf{Z}^n$ by fixing $x(\vec{v}) = 0$ if $\vec{v} \notin D$.

Operations on real numbers can be "lifted" to signals in a natural way. In particular, if x and y are nD-signals and $\lambda \in \mathbf{R}$ is a scalar constant, we define addition and scalar multiplication of signals as:

$$(x + y)(\vec{u}) = x(\vec{u}) + y(\vec{u}) \qquad \text{and} \qquad (\lambda x)(\vec{u}) = \lambda(x(\vec{u})).$$

Equipped with these two operations, $\mathbf{Z}^n \to \mathbf{R}$ is a vector space over $\mathbf{R}$.

Another importation operation is the *shifting* of a signal by a constant offset. Let x be an nD-signal and $\vec{u} \in \mathbf{Z}^n$ a vector of coordinates. The translation of x by $\vec{u}$, denoted $\tau_{\vec{u}}(x)$, is defined as:

$$\tau_{\vec{u}}(x) = x \circ \Delta_{\vec{u}}$$

where $\Delta_{\vec{u}}$ is the translation of coordinates: $\vec{v} \mapsto \vec{v} + \vec{u}$. In other words,

$$\tau_{\vec{u}}(x)(\vec{v}) = x(\vec{v} + \vec{u}).$$

3.3.2 Linear Shift-Invariant Systems

A multidimensional system (or filter) T is a dimension-preserving transformation between signals: it transforms an n-dimensional input into an n-dimensional output. It is thus a map $T : (\mathbf{Z}^n \to \mathbf{R}) \to (\mathbf{Z}^n \to \mathbf{R})$. T is called *Linear, Shift-Invariant* (LSI) if it verifies the following properties:

Linearity For any scalar $k \in \mathbf{R}$ and any inputs $a, b \in (\mathbf{Z}^n \to \mathbf{R})$:

$$T(\lambda a + b) = \lambda T(a) + T(b).$$

Shift-Invariance For any vector $\vec{u} \in \mathbf{Z}^n$,

$$T(\tau_{\vec{u}}(x)) = \tau_{\vec{u}}(T(x)),$$

The linearity requirement implies that T is a linear map in the usual sense, i.e., preserves the vector space structure. Shift-invariance means that it also preserves shifts: applying a shift *before* or *after* T produces the same result. In other words, LSI filters commute with translation of signals.

Examples

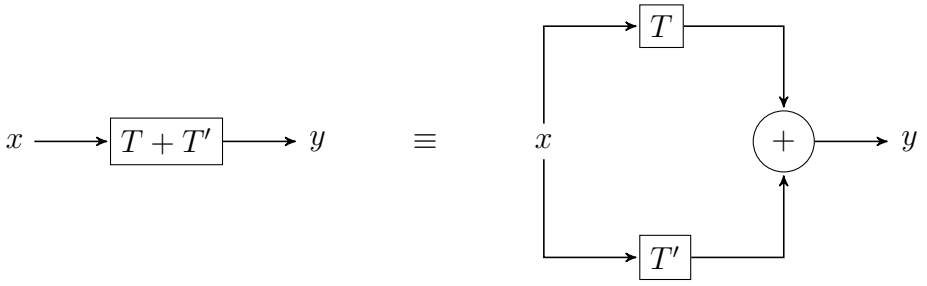
- Any LTI system, such as the FIR filter, is also LSI.
- The Deriche filter, like many image processing algorithms, is a two-dimensional LSI system. This is a sensible property for an edge detector: indeed, the gradient operator $\vec{\nabla}$ is linear and edge localization should be translation-invariant.

3.3.3 Algebraic Structure of LSI Systems

In Section 3.2, we saw that the 2D Deriche filter is made of compositions and summations of simpler 1D LTI filters, operating over rows and columns in both directions. LTI and LSI systems are *stable* over linear operations and composition, which guarantees that the Deriche kernel is indeed linear and shift-invariant.

More precisely, let T, T' be two filters of same dimensionality n, x be an nD-signal, λ a scalar value and $\vec{d}$ a vector of $\mathbf{Z}^n$. We define the following operations:

Addition $(T + T')(x) \equiv T(x) + T'(x)$.



Multiplication by a scalar $(\lambda T)(x) \equiv \lambda\,(T(x))$.

Composition $(T' \circ T)(x) = T'(T(x))$.

Note that these operations are *not* the same as those defined for signals: they map systems to systems, and not signals to signals.

One easily proves that the space of LSI filters is stable (or *closed*) under these operations, i.e., their result is always linear and shift-invariant. Moreover, composition is a bilinear operation:

$$(\lambda x + y) \circ z = \lambda(x \circ z) + y \circ z \qquad \text{and :} \qquad z \circ (\lambda x + y) = \lambda(z \circ x) + z \circ y$$

As a direct consequence, observe by setting $\lambda = 1$ that composition *distributes* over addition. Finally, the "empty" system:

$$\mathrm{id}_n : x \mapsto x$$

is an identity element with respect to composition. These properties equip LSI systems with a structure of unital algebra over $\mathbf{R}$, *i.e.*, a vector space with a bilinear product and a multiplicative identity.

3.3.4 Impulse Response

An LSI system is characterized by its impulse reponse, i.e., the output of the system to a unit impulse. A n-dimensional unit impulse is an nD-signal defined as:

$$\delta_n(\vec{x}) = \begin{cases} 1 & \vec{x} = \vec{0} \\ 0 & \text{otherwise} \end{cases}.$$

Let $h_T = T(\delta_n)$ (or just h, where T is clear from context) denote the impulse response of T. It can be shown that for any signal x,

$$T(x) = h_T * x,$$

where $*$ denotes the convolution operation:

$$(x * y)(\vec{v}) = \sum_{\vec{w} \in \mathbf{Z}^n} x(\vec{v} - \vec{w}) y(\vec{w})$$

Examples

- The impulse response of a FIR filter is given by its coefficients. Indeed,

$$h(n) = \sum_{i=0}^{N} b_i \delta_n(n - i) = \begin{cases} b_n & 0 \leq n \leq N \\ 0 & otherwise \end{cases}$$

  In other words, a FIR filter computes the convolution of its input with its coefficients.

- In general, the impulse reponse of an n-dimensional *recursive* LSI system is an n-dimensional surface with infinite support. For example, the truncated impulse response of the Deriche filter, centered around the origin, is shown in Figure 3.1. This impulse response corresponds to the mask that should be used in a non-recursive implementation to approximate the same frequency response.

3.3.5 Transfer Function and $\mathcal{Z}$-Transform

Impulse responses describe the behavior of systems in the spatial (or temporal) domain. For analysis purposes, though, the *frequency*-domain point-of-view is often more convenient. The frequency behavior of LSI systems can be captured through *transfer functions*.

The transfer function of a system is the result of taking the $\mathcal{Z}$-transform of its impulse response. The $\mathcal{Z}$-transform is the discrete analog of the Laplace transform. It takes a signal from the spatial domain to the complex frequency domain, and is defined as:

$$\mathcal{Z}(x)(\vec{z}) = \sum_{\vec{w} \in \mathbf{Z}^n} x(\vec{w}) z_1^{-w_1} \dots z_n^{-w_n},$$

where $\vec{z} = (z_1, \dots, z_n) \in \mathbf{C}^n$.

In practice, this definition is rarely useful to compute transfer functions, thanks to the following properties:

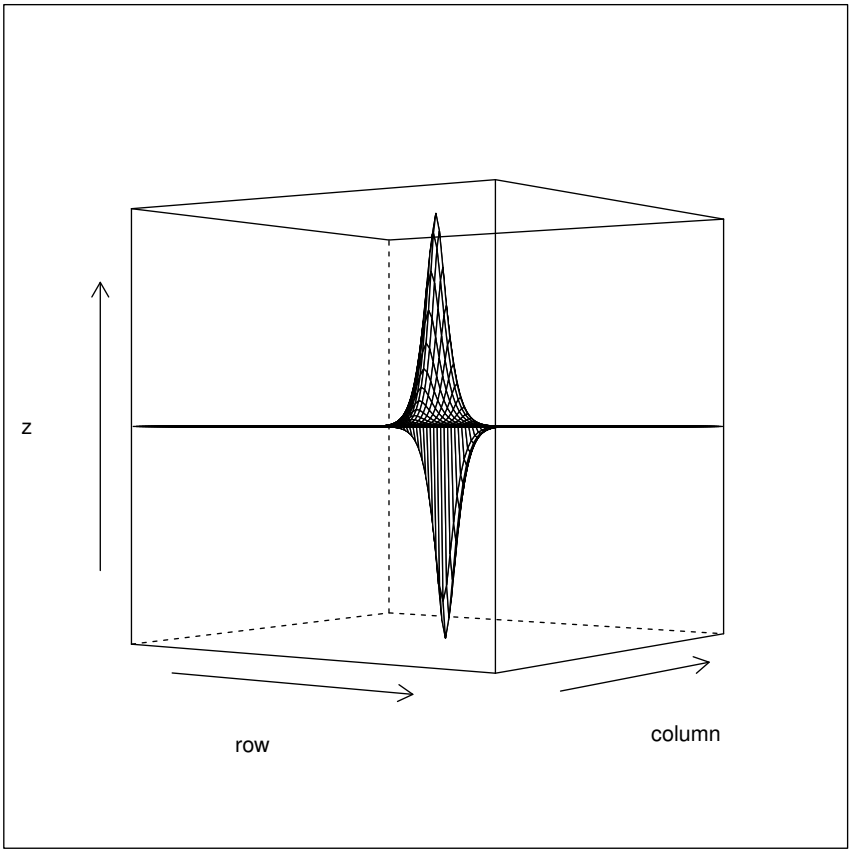


Figure 3.1: Impulse Response of the deriche edge detector estimated by our tool (horizontal gradient).

Linearity For any $\lambda \in \mathbf{R}$ and any signals x and y,

$$\mathcal{Z}(\lambda x + y) = \lambda \mathcal{Z}(x) + \mathcal{Z}(y).$$

Space shifting For any vector $\vec{u} = (u_1, \ldots, u_n)$,

$$\mathcal{Z}(x - \vec{u})(\vec{z}) = z_1^{-u_1} \ldots z_n^{-u_n} \mathcal{Z}(x)(\vec{z}).$$

Spatial convolution / frequential product The $\mathcal{Z}$-transforms maps spatial convolutions to products in the frequency domain:

$$\mathcal{Z}(x * y) = \mathcal{Z}(x)\mathcal{Z}(y).$$

These results allow us to quickly determine the transfer function of a composite system from that of its sub-parts.

Example: The FIR Filter Recall that the impulse response of an FIR filter is given by its coefficients. It may be written as a linear combination of unit impulses:

$$h_T = \sum_{i=0}^{N} b_i \tau_{-i}(\delta)$$

Knowing that $\mathcal{Z}(\delta) = 1$ and applying the above properties, we have:

$$\mathcal{Z}(h_T)(z) = \sum_{i=0}^{N} b_i z^{-i}.$$

Example: IIR Filter An IIR filter is usually defined by a recurrence equation of form:

$$y(n) = \sum_{i=0}^{N} b_i x(n-i) - \sum_{j=1}^{M} a_j y(n-j) \quad \Leftrightarrow \quad y(n) + \sum_{j=1}^{M} a_j y(n-j) = \sum_{i=0}^{N} b_i x(n-j).$$

With the same kind of reasoning as for the FIR filter, we can compute its transfer function:

$$z \mapsto \frac{\sum_{i=0}^{N} b_i z^{-i}}{1 + \sum_{j=1}^{M} a_j z^{-j}}$$

3.3.6 Frequency Response

The frequency response of the system is computed from the transfer function by constraining each component of its input vector to lie on the unit circle:

$$H_T(\omega_1, \ldots, \omega_n) \equiv H_{z,T}(e^{i\omega_n}, \ldots, e^{i\omega_n}).$$

where ω_1 and ω_2 are real numbers. We have the following, fundamental identity:

$$H_T = \mathcal{F}(h_T),$$

where $\mathcal{F}$ denotes the Fourier transform.

The relation between the impulse response, the transfer function and the frequency response of a system is pictured in Figure 3.2. Remark that, even though H_T only has real parameters, it does not contain less information on the frequency behavior than $H_{z,T}$. In fact, $H_{z,T}$ can be retrieved from H_T using the following relation:

$$H_{z,T} = \mathcal{Z}(\mathcal{F}^{-1}(H_T)).$$

3.4 Analytical Accuracy Evaluation for LSI Systems

We now present our approach to derive analytical accuracy models from source-code description of multi-dimensional LSI systems.

3.4.1 Overview

Our approach can be decomposed into four steps.

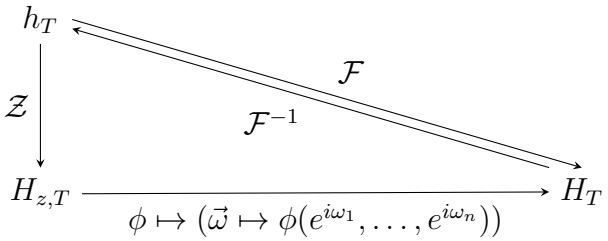


Figure 3.2: Relation between the impulse response, transfer function and frequency response of a system T. The transfer function $H_{z,T}$ is the $\mathcal{Z}$-transform of the impulse response. The frequency response H_T is obtained by restricting the transfer function to the unit circle for each component. Finally, h_T and H_T are related by the Fourier transform.

Representation Extraction

The first step of our method is to extract a multi-dimensional flowgraph representation from an algorithmic (e.g., C/C++) specification. Our technique is based on the polyhedral model, a mathematical framework for analysis and transformation of programs with regular control flow and access patterns. The source code is analyzed and translated into an equivalent system of recurrence equations, which is then further refined into a flowgraph.

Coefficients Quantization

The effect of coefficients quantization is assessed by comparing the frequency responses of the floating-point and fixed-point implementations. The frequency responses are computed automatically.

Accuracy Model Construction

Once a set of fixed-point coefficients compatible with application requirements has been determined, an analytical accuracy model is derived from the flowgraph representation. This is done by computing the transfer function from each node to the output, modeling the impact of quantization noise on the final result.

Wordlength Optimization

The accuracy model constructed in the previous step can then be exploited to perform fast automatic wordlength optimization.

3.4.2 Multidimensional Flow-Graphs

Multidimensional LSI systems can be conveniently represented as *multidimensional flowgraphs* (MDFGs). As an example, consider the following subset of the Deriche

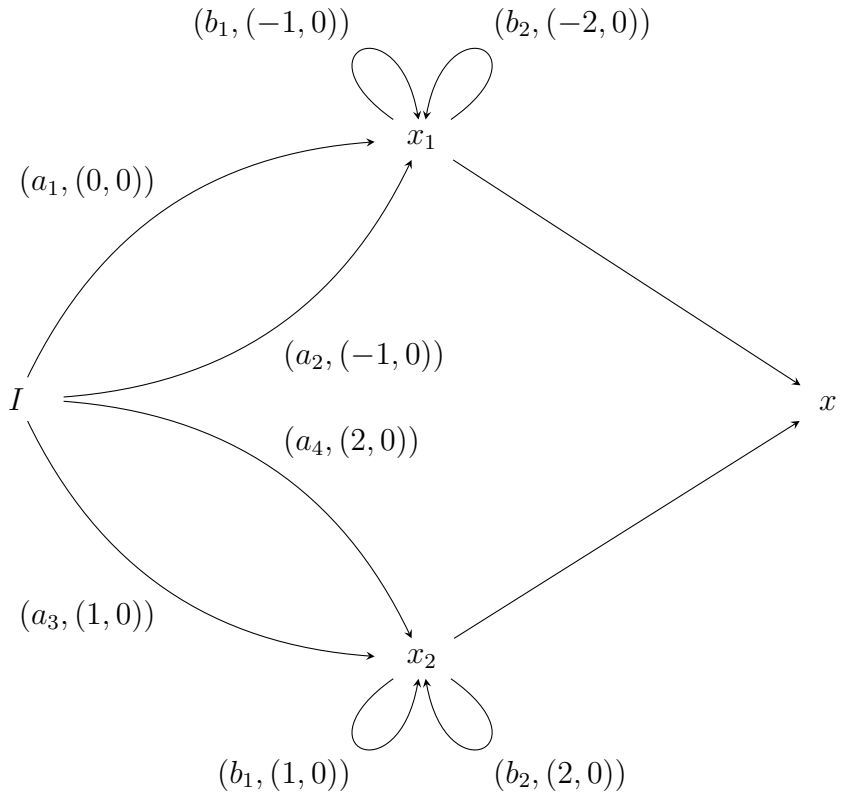


Figure 3.3: Partial flowgraph of the Deriche filter

filter:

$$\begin{aligned} x_1(i,j) &= a_1 I(i,j) && + a_2 I(i-1,j) + b1 x_1(i-1,j) && + b_2 x_1(i-2,j) \\ x_2(i,j) &= a_3 I(i+1,j) && + a_4 I(i+2,j) + b1 x_2(i+1,j) && + b_2 x_2(i+2,j) \\ x &= x_1 + x_2 \end{aligned}$$

The corresponding flowgraph is shown in Figure 3.3. Each node in the graph represents an equation (or the input signal I). Each equation is the summation of incoming edges, labeled with a multiplier and an offset vector. For example, The edge $(a_3, (1, 0))$ from I to x_2 represents the summand $a_3 I(i + 1, j)$ in the definition of x_2.

Linear SFGs can be trivially encoded as MDFGs. For example, delay nodes may by replaced as node with incoming edge labeled: $(1, -1)$, where 1 represents the unit multiplier and -1 the time offset.

3.4.3 Inference of MDFGs from Source Code

The first step in our approach is to extract the flow-graph representation of a system from its algorithmic specification. It is based on *Systems of Affine Recurrence Equations* (SAREs), a denotational representation of programs than can be inferred from polyhedral source code.

Polyhedral Model

The polyhedral model is a framework for modeling, analyzing and transforming a large class of regular programs and program fragments known as *Static Control Parts* (SCoPs) or *Affine Control Loops* (ACLs). While several extensions to the model have been proposed [24, 25], SCoPs are usually defined by the following constraints:

- The program is composed exclusively of for loops, if-then-else statements and computations on scalar values and array elements.
- Guards and array indices take the form of affine constraints on enclosing loop iterators, statically-known constants and symbolic parameters.

A consequence of this definition is that control flow is *static*, as guards cannot refer to values computed within the same SCoP. In other words, the sequence of instructions is determined exactly by parameter values.

Example The following program is a SCoP.

```
float sum=0;
for (int i=0; i<N; i++) {
  sum += arr[i];
}
```

Example This program is *not* a SCoP: control flow depends on values of array elements and is thus not static.

```
float sum=0;
for (int i=0; i<N; i++) {
  if (arr[i] > 0)
    sum += arr[i];
}
```

Example This program is not a SCoP either: control flow *is* static, but guards contain non-affine operations.

```
float sum=0;
for (int i=0; i<N; i++) {
  for (int j=0; j<i*i; j++) {
    sum += arr[i];
  }
}
```

Array Dataflow Analysis

The flow of computations in polyhedral programs can be determined exactly using a powerful dependence analysis technique called *Array Dataflow Analysis* (ADA) [26]. It captures *instance-wise* and *element-wise* dependences:

- Instance-wise means that the successive iterations of each statement in the program are distinguished from each other.
- Element-wise means that accesses to different elements from the same array are also distinguished.

As a simple example, consider the following fragment:

```
for (int i=0; i<N; i++)
S0:  x[i] = 0;
     for (int j=0; j<N; j++)
S1:      x[i] = x[i] + a[j];
```

Without instance-wise information, all that can be said is that `S1` depends on both `S0` and `S1`. Without element-wise information, an instance of `S1` at (i, j) depends on all instances of `S0` and `S1`, as they all write the same variable `x`. With ADA, though, the dependence information found for the read `x[i]` in `S1` is the following:

$$\text{Producer of } \texttt{x[i]} \text{ at } \texttt{S1}(i,j) = \begin{cases} \texttt{S0}(i) & j = 0 \\ \texttt{S1}(i, j-1) & j > 0 \end{cases}.$$

For each operand used in each statement instance, we can thus determine *exactly* the statement instance (if any) that produced this value.

From ADA to SAREs

The dataflow information given by ADA can be used to transform the program to a semantically equivalent system of affine recurrence equations. Each statement in the program becomes an equation, where references to array elements are replaced by the producer of the corresponding value. For example, the program above can be written:

$$\begin{cases} S_0(i) = 0 \\ S_1(i,j) = a(j) + \begin{cases} S_0(i) & j = 0 \\ S_1(i, j-1) & j > 0 \end{cases} \end{cases}$$

This form captures computations and data dependences in a single representation, abstracting away storage locations.

From SAREs to Multidimensional Flow-Graphs

While multidimensional flow-graphs can always be seen as systems of recurrence equations, the opposite is not necessarily true. Moreover, even when that is the

case, additional work is sometimes required to exhibit the underlying flow-graph nature. This is almost always the case for SAREs built from ADA.

For example, consider the following code, representing an FIR filter:

```
float tmp[3];

for (int i=0; i<N; i++) {
S0: tmp[2] = x[i];
S1: y[i] = 0.25*tmp[0] + 0.5*tmp[1] + 0.25*tmp[2];

    for (int j=0; j<2; j++)
S2:     tmp[j] = tmp[j+1];
}
```

A simplified version of the output obtained after ADA and SARE extraction is shown below:

$$\begin{cases} S_0(i) = x(i) \\ S_1(i) = 0.25 * S_2(i-1,0) + 0.5 * S_2(i-1,1) + 0.25 * S_0(i) \\ S_2(i,j) = \begin{cases} S_0(i) & j = 1 \\ S_2(i-1,j+1) & \text{otherwise} \end{cases} \end{cases}$$

Even though FIR filters are LSI systems, the system above is not directly equivalent to a flow-graph. The reason is that equations and operands do not all have the same dimensionality: S_0 and S_1 are one-dimensional while S_2 has two dimensions because of the copy loop.

The solution here is to *inline* the definition of S_2 in all use sites. However, it cannot be done directly because of the recursive reference in S_2. We tackle this problem by computing the transitive closure of the copy relation defined by S_2. The equation changes to:

$$S_2(i,j) = S_0(i-j+1)$$

We can now inline S_2 and the system becomes:

$$\begin{cases} S_0(i) = x(i) \\ S_1(i) = 0.25 * S_0(i-2) + 0.5 * S_0(i-1) + 0.25 * S_0(i) \end{cases}$$

Finally, we can also inline S_0 into S_1, which gives us the following single equation:

$$S_1(i) = 0.25 * x(i-2) + 0.5 * x(i-1) + 0.25 * x(i).$$

Another common, more subtle difficulty arises because index dimensions in SAREs extracted from source code correspond to *iteration* dimensions and not *data* dimensions. To illustrate this problem, consider the simplified Deriche filter code below:

```
// Horizontal (left-to-right) pass
for (int i=0; i<W; i++) {
  ym1=0; ym2=0; xm1=0;
  for (int j=0; j<H; j++) {
```

```
S1:   x1[i][j] = a1*I[i][j] + a2*xm1 + b1*ym1 + b2*ym2;
S2:   xm1 = I[i][j];
S3:   ym2 = ym1;
S4:   ym1 = x1[i][j];
  }
}

// Right-to-left pass not shown for brevity.
for (int i=0; i<W; i++) {
  ...
}

// x = x1+x2
for (int i=0; i<W; i++)
  for (int j=0; j<H; j++)
S5:   x[i][j] = x1[i][j] + x2[i][j];

// Vertical (top-to-bottom) pass not shown for brevity.
for (int j=0; j<H; j++) {
  ...
}

// Vertical (bottom-up) pass
for (int j=0; j<H; j++) {
  tp1=0; tp2=0; yp1=0; yp2=0;
  for (int i=W-1; i>=0; i--) {
S6:   y2[i][j] = a7*tp1 + a8*tp2 + b1*yp1 + b2*yp2;

S7:   tp2 = tp1;
S8:   tp1 = x[i][j];
S9:   yp2 = yp1;
S10:  yp1 = y2[i][j];
  }
}


// y = y1+y2
for (int i=0; i<W; i++)
  for (int j=0; j<H; j++)
S5:   y[i][j] = y1[i][j] + y2[i][j];
```

In this implementation, the horizontal passes visit columns of the image in the inner loop, whereas the vertical passes visit columns of the image (after horizontal filter) in the *outer* loop. This can be viewed as applying a horizontal pass on the transpose of the image. In fact, the code for horizontal and vertical passes are almost identical due to this transposed view of the image.

Ignoring boundary conditions, the SARE inferred for the above program is:

$$\begin{cases} S_1(i,j) = a_1 I(i,j) + a_2 S_2(i,j-1) + b_1 S_4(i,j-1) + b_2 S_3(i,j-1) \\ S_2(i,j) = I(i,j) \\ S_3(i,j) = S_4(i,j-1) \\ S_4(i,j) = S_1(i,j) \\ S_5(i,j) = S_1(i,j) + x_2(i,j) \\ S_6(j,i) = a_7 S_8(j,i+1) + a_8 S_7(j,i+2) + b_1 S_{10}(j,i+1) + b_2 S_9(j,i+1) \\ S_7(j,i) = S_8(j,i+1) \\ S_8(j,i) = S_5(i,j) \\ S_9(j,i) = S_{10}(j,i+1) \\ S_{10}(j,i) = S_6(j,i) \\ S_{11}(i,j) = y_1(i,j) + S_6(j,i) \end{cases}$$

Because ADA constructs statement domains by examining the loop structure – and not array dimensions – statements corresponding to the horizontal passes $(S_1, \ldots, S_4)$ and array summations (S_5, S_{11}) are defined on (i,j) coordinates, whereas vertical passes $(S_6, \ldots, S_{10})$ are defined on transposed (j,i) coordinates. The link between the two views is made in equation S_8 and S_{11}. To transform a SARE into a MDFG, we actually need to turn it into a system of *uniform* recurrence equations (SURE), such that dependence patterns are reflected by constant dependence vectors. Such is not the case here: for example, the dependence vector between S_5 and S_8 in the definition of S_8 is:

$$(i-j, j-i)$$

which is clearly non-constant.

However, since the non-uniformity is introduced by benign permutations of dimensions, the SARE can be transformed into an equivalent SURE. We adapt techniques for uniformization/localization [27, 28] of dependences to properly align the equations. Simple pattern-matching can then be used to retrieve the constant coefficients and build a SURE/MDFG. If the system cannot be properly uniformized, then the analysis fails and no MDFG can be built.

We voluntarily left aside the problem of boundary conditions. They are reflected in the system as additional case branches. For example, the true equation for S_1 is equivalent to:

$$S_1(i,j) = \begin{cases} a_1 I(i,j) + a_2 S_2(i,j-1) + b_1 S_4(i,j-1) + b_2 S_3(i,j-1) & j > 0 \\ a_1 I(i,j) & j = 0 \end{cases}$$

We use ad-hoc heuristics on domains and conditions to select the case-branch that corresponds to the general case. More sophisticated algorithmic-template recognition methods [29,30] could be used to determine that other case branches correspond to 0-initial conditions outside the image domain.

3.4.4 Computation of Transfer Functions

A key enabler to analyze the quantization of coefficients and construct the accuracy model is the ability to derive transfer functions from an MDFG. In this section, we present a recursive algorithm computing the transfer function from each node in the graph to the output. The result is a map associating each node in the graph to the transfer function representing the propagation of its contribution to the final result.

The idea behind our algorithm is the following. Let v_i be a node in the graph and $(v_j, k_j, \vec{d_j})$ the set of $(\text{source}, \text{multiplier}, \text{offset})$ triples representing incoming edges. We can distinguish two cases:

- **Case 1:** Node v_i does not belong to any cycle. Then, for each predecessor v_j and any node v_k, let $TF_{k\to j}$ be the transfer function from v_k to v_j. By simple applications of the rules in Section 3.3.5, we have:

$$TF_{k\to i} = \sum_j k_j z_1^{d_{j,1}} \dots z_n^{d_{j,n}} TF_{k\to j}.$$

- **Case 2:** There is at least one cycle involving v_i. Then, let $v_{i'}$ be a dummy node replacing v_i as the source of every edge going out of v_i, thus breaking any cycle. We have:

$$TF_{k\to i} = \frac{\sum_j k_j z_1^{d_{j,1}} \dots z_n^{d_{j,n}} TF_{k\to j}}{1 - \sum_j k_j z_1^{d_{j,1}} \dots z_n^{d_{j,n}} TF_{i'\to j}}.$$

In other words, in the absence of any cycle, transfer functions can be computed by simple application of the computation rules given in Section 3.3.5. Cycles are eliminated simply by considering the current node as an input node and then solving for the actual transfer function by including recursive contributions. A pseudocode description of the algorithm is given in Figure 1.

By construction, we observe that no transfer function between any pair of nodes is computed twice. We conclude that this algorithm is of quadratic complexity $O(n^2)$ where n represents the number of vertices in the graph.

This algorithm is fundamentally similar to the one presented by Menard et al. [23] for LTI systems: likewise, we dismantle cycles to recursively solve for the global transfer function. However, their technique was much more complex, requiring 4 graph transformations, enumeration techniques to break down the graph into cycles and explicit substitutions to compute the transfer function. Our algorithm simply relies on a memoization strategy in the depth-first search traversal to resolve circular references and guarantee termination.

3.4.5 Quantization of Coefficients

Word-Length Optimization is essentially concerned with quantization of signals: input values and intermediary values. The quantization error of constant system coefficients, such as the coefficients of the FIR or Deriche filters, does not vary over time/space and does hence not lend itself well to statistical analysis and accuracy

Function *computeTFs(output: Node): Map<Node,TF>* **is**
  aux(node, $\emptyset$)
end

Function *aux(node: Node, baseNodes: Set<Node>): Map<Node, TF>* **is**
  `/* Base case.  If the node is a base node, return identity. */`
  if *node* $\in$ *baseNodes* **then**
    return {node $\rightarrow$ 1};
  end
  `/* Construct transfer function map from incoming edges.  */`
  tmpMap := {s $\rightarrow$ 0};
  foreach $(pred, k, \vec{d}) \in$ *incomingEdges(node)* **do**
    map' := aux(pred, baseNodes $\cup$ {node});
    S := $kz_1^{d_1} \ldots z_n^{d_n}$; `/* Transfer function factor.  */`
    foreach *other* $\in$ *keys*(*map*$'$) **do**
      if *other* $\neq$ *node* **then**
        tmpMap[other] += S $\times$ map'[other];
      end
    end
  end
  `/* Remove recursive contribution.  */`
  selfTF := tmpMap[node]; `/* 0 if there is no cycle.  */`
  map := {};
  foreach *other* $\in$ *keys(tmpMap)* **do**
    if *other* $\neq$ *node* **then**
      map[other] += $\frac{\text{tmpMap[other]}}{1-\text{selfTF}}$;
    end
  end
  return map;
end

Algorithm 1: Computation of transfer functions

characterization. However, these errors affect the response of the system and aggressive quantizations can naturally compromise its function. It is usually assumed that these problems have been handled prior to floating-point to fixed-point conversion, for example by assessing the sensitivity of the system transfer function to small variations of coefficients. Instead, we propose to exploit the knowledge gathered during the analysis of the system to help the designer verify that the frequency characteristics of the filter with quantized coefficients matches functional requirements.

These requirements are usually expressed as a *frequency response template*. Typically, min and max-bounds are defined on the frequency response of the system. However, inspection by an expert designer is often required. Since we can already compute the transfer function of the quantized system, we propose to compute and display the frequency response, using the relation:

$$H(\omega_1, \ldots, \omega_n) = H_z(e^{i\omega_1}, \ldots, e^{i\omega_n}).$$

If possible, automatic validation of frequency response based on a specified template can also be performed. Integrating the quantization of coefficients into the scope of computed-assisted floating-point to fixed-point conversion provides benefits in terms of productivity and leaves less room for errors.

3.4.6 Accuracy Model Construction

The propagation of quantization noise from each node in the graph is fully determined by the coefficients of the impulse response associated to that node. Indeed, let e_x be the error signal at node x and e'_x its propagation to the output. As we are dealing with LSI systems, the resulting error at the output is given by:

$$e'_x = h_{x,T} * e_x$$

Following the PQN-model, we represent the values of e_x over space as independent, identically distributed (i.i.d) random variables. Let μ_{e_x} and $\sigma^2_{e_x}$ stand respectively for the mean and variance of the underlying probability distribution. We have:

$$e'_x(\vec{v}) = \sum_{\vec{w}} h_{x,T}(\vec{w}) e_x(\vec{v} - \vec{w}).$$

Because of non-correlation we conclude that:

$$\mu_{e'_x} = \mu_{e_x} \sum_{\vec{v}} h_{x,T}(\vec{v}) \qquad \sigma^2_{e'_x} = \sigma^2_{e_x} \sum_{\vec{v}} h^2_{x,T}(\vec{v})$$

To compute the first two moments of e'_x, we thus need to determine the sum and sum of squares of the impulse response coefficients. We will show two different ways to achieve this: a direct one, using the flowgraph representation to approximate the impulse response by abstract simulation on a unit impulse ; and a slightly more efficient approach based on the frequency response.

Direct approach A *well-formed* [31] SURE / MDFG gives a computable specification of a system. In our context, the SURE are derived from the schedule of a program. This property ensures that the resulting recurrence equations are indeed computable. We exploit this fact to simulate the subsystem corresponding to the propagation of each quantization noise: we use a unit impulse as input to compute the impulse response coefficients over a sufficiently large window around the origin, naively computing the values of each equation "on-demand".

This direct simulation scheme requires the introduction of boundary conditions for each equation. We assume that the filters (and sub-filters) we study are *stable*:

$$M = \sum_{\vec{v}} |h_T(\vec{v})| < \infty.$$

If the input is bounded by B, then the output is bounded by MB: the system cannot result in infinite amplification of inputs. This is a reasonable assumption in almost all applications. A direct consequence is that:

$$\lim_{|\vec{v}|\to\infty} h_T(\vec{v}) = 0.$$

We exploit this fact by setting $x(\vec{v}) = 0$ for any equation x when $|\vec{v}| > r$. This is a good approximation provided r is large enough, and allows for the computation of the impulse responses to terminate. However, determining the correct value of r such that the approximation error is below some bound ε is not trivial. In our experiments, though, we found values of $r = 50$ to give excellent results.

Approach based on the frequency response An alternative approach to the direct solution above is to compute the frequency response of the system based on the formulas in Section 3.3. The sums $\sum h(\vec{v})$ and $\sum h^2(\vec{v})$ can also be interpreted in the frequency domain:

- The first one is simply the gain of the system for a constant input, i.e., the frequency response at $\vec{0}$:
$$\sum_{\vec{v}} h(\vec{v}) = H(\vec{0})$$
.
- The second one is the L^2 norm of the impulse response, and is preserved in the frequency domain. According to Parseval's theorem:
$$\sum_{\vec{v}} h^2(\vec{v}) \approx \frac{1}{N} \sum_{\vec{w}} H^2(\vec{w}).$$

where N denotes the number of samples over $(-\pi, \pi]^n$ taken for the frequency response. The method above is easy to implement given the transfer functions of the system and is slightly more efficient than the direct simulation-based approach, as a closed-form expression of the frequency response can be obtained without having to unroll the recurrence equations.

Table 3.1: Model construction time for our tool and ID.Fix.

Algorithm	ID.Fix (s)	Our tool (s)
IIR8	23.1	20.5
Sobel (32×32)	169.1	9.2
Sobel (64×64)	2173.1	9.7
Sobel (128×128)	-	9.4
Sobel (32×32)	160.1	9.2
Sobel (64×64)	2010.9	9.5
Sobel (128×128)	-	9.4
Deriche blur (16×16)	-	6.5

3.5 Experimental Validation

We integrated our approach, using the direct (time-domain) variant, within a compiler framework. The front-end (SURE extraction) was implemented within the GeCoS [2] flow, developed at Irisa. The backend was written in OCaml.

In Table 3.1, we contrast the scalability of our approach to that ID.Fix, an accuracy analysis tool based on the work Ménard et al. [23]. The lesson we draw from these experiments is that, while our tool is insensitive to problem size when dealing with image filters, ID.Fix is not and suffers from major scalability problems with larger problem sizes. In fact, at the time we ran these experiments, ID.Fix could not handle filters over images larger than 64x64, even for non-recursive filters.

ID.Fix has seen major engineering efforts and it is possible that performance has improved since then. In particular, for the non-recursive Sobel and Gaussian blur filters, a simple analysis shows that run time should be proportional to the number of pixels in the image and thus only increase by a factor of four between the 32×32 and 64×64 versions, In practice, though, we observed a performance degration of more than 13 times. This suggests the presence of performance bugs. In any case, the story stays the same: our tool can scale to any (or even parametric) image size because it captures the dimensionality nature of these filters, whereas ID.Fix cannot, as it needs to "flatten" the control flow to treat these algorithms as 1D kernels.

In a second round of experiments, we compared the error predicted by our tool for a set of fixed-point configurations with the one actually observed between fixed-point and floating-point simulations. We emphasize that, unlike many prior work, these experiments have been run against real data (sound and image files) and not randomly generated test benches. It means that we also take into account potential estimation errors due to correlation between noises. This is important as most real world data shows strong spatial correlation, unlike artificial data where each sample has been randomly generated. Our results show that these phenomena do not strongly affect the validity of the Widrow hypothesis, as the deviations we observed were all below 5%.

[2] `http://gecos.gforge.inria.fr`

Table 3.2: Validation of our model against simulations and ID.Fix.

Algorithm	Actual error (dB)	Predicted error	Estimation error (dB/%)
IIR8	-17.80	-17.84	-0.04/-0.2%
Sobel	11.62	12.04	0.42/3.6%
Gauss	3.78	3.78	<0.01/0.1%
Deriche	-18.01	-18.06	-0.05/-2.78%

3.6 Future Work and Extensions

In Chapter 2, we saw that extensions of analytical methods to non-LTI methods are almost all-based on a linearization of the system with time-varying coefficients. This linearization, based on perturbation theory, is reflected as a graph transformation where noise propagation is modeled as linear operations on error signals with input coefficients. The noise model is usually built from coefficient-signals characteristics, which are assessed with varying numbers of parameterization simulations.

Although we haven't studied this topic in much detail, we believe this approach can also be applied to multidimensional algorithms. However, to properly estimate signal statistics, the number of required samples increases exponentially with the number of dimensions (this phenomenon is sometimes called *the dimensionality curse*). It is possible that, for systems of high-dimensionality, the amount of memory and/or time required to perform the parameterization phase becomes prohibitive.

An obvious limitation of our approach is the applicability to polyhedral programs only. We cannot compactly capture regular but non-affine control flow as in the FFT transform: in such cases, parts of the programs must still be unrolled until the polyhedral conditions are met. Abstract models based on probabilistic semantics may be constructed to conservatively capture the propagation of noise probability distributions through arbitrary control flow. Some work has been done in this direction [32] based on discretization of the measure space. However, the accuracy and performance of the technique are tied to the granularity of that discretization - to our knowledge, no purely analytical lattice has been defined.

Even within the polyhedral model, we expect the round-off error behavior of some algorithmic patterns to be difficult to capture. For example, in the next two chapters, we will consider the hardware implementations of *iterative stencil computations* (or stencils for short). Let $G \subset \mathbf{Z}^n$ denote an n-dimensional domain. A *Jacobi-style* stencil is given by a transformation $(G \to \mathbf{K}) \to (G \to \mathbf{K})$, defined as a recurrence relation of form:

$$D_{n+1}(\vec{w}) = f(\vec{w}, D_n(\vec{w} + \vec{d_0}), \ldots, D_n(\vec{w} + \vec{d_m}))$$

In most applications, one is interested, given some initial state D_0, in computing D_T where T may be a constant or computation-dependent number of iterations.

Note that, in the general case, the computation may depend on spatial coordinates $\vec{w}$. For example, $\vec{w}$ could be used to implement absorbing boundary conditions, or the coefficients actually depend on the spatial position. the iteration number. In simpler cases, such as the heat equation with uniform coefficients, the recurrence

relation is defined as a dot product with a vector of constant coefficients:

$$D_{n+1}(\vec{w}) = \left(D_n(\vec{w} + \vec{d_0}) \ \ldots \ D_n(\vec{w} + \vec{d_m}) \right) \cdot \begin{pmatrix} k_0 \\ \vdots \\ k_m \end{pmatrix}$$

In such cases, assuming boundary conditions $D_n(\vec{w}) = 0$ if $\vec{w} \notin G$, we may model the effect of a single stencil iteration as the convolution of the grid with a mask of coefficients, which is a linear filter with transfer function H_z. Then the stencil as a whole may be modeled as a linear filter with transfer function : H_z^T. If T is a known constant, the techniques presented in this chapter may be applied, with a caveat: signal power over successive iterations may be reduced such that Widrow's quantization theorem does not apply anymore, thus breaking a fundamental assumption of our modeling. Stencil with space-varying coefficients and non-constant coefficients are even harder to model. In the general case, assessing the accuracy of iterative computations may require other sets of techniques.

3.7 Conclusion

Over the last two chapters, we discussed trade-offs opportunities between, e.g., silicon area and *accuracy*. In Chapter 2, we exposed the problem *accuracy evaluation* and the limitations of simulation-based methods. In Chapter 3, we presented our contributions to *analytical* accuracy evaluation, a class of methods that seek to enable more thorough design space exploration through accuracy models. Our work generalizes earlier techniques, applicable to linear systems, to the multidimensional case. To accomodate a source-level design flow, this generalization required the use of techiques from the polyhedral compilation toolset.

In the remaining of this thesis, we focus on another class of algorithms: iterative stencil computations. Due to their ubiquity, stencils and their acceleration have been the subject of much study. The diversity of their applications and the large number of corresponding algorithms gives rise to different constraint sets that each call for specific trade-offs between throughput, bandwidth requirements and area, to name a few. After a review of generic techniques for the optimization of stencils in Chapter 4, we will present our results and contributions in Chapter 5, based once again on a rigorous modeling of the design space.

Chapter 4

Implementation and Optimization of Stencil Computations

4.1 Introduction

Iterative stencil computations (often just called *stencils*) are a family of regular algorithms used in application domains as varied as numerical analysis, computer simulations and image processing. Stencils operate over multidimensional grids of data, repeatedly updating each cell from neighbor values in successive timesteps. This computational pattern can be easily described as compact loop nests, as in Figure 4.1, where the outer loop iterates over time iterations while the innermost loops scan the spatial grid.

```
a[0][_] = ...; // Set initial conditions.

for (int t=1; t<T; t++) // Temporal loop
  for (int x=1; x<N+1; x++) // Spatial loop
    a[t][x] = (a[t-1][x-1]+a[t-1][x]+a[t-1][x+1])/3;
```

Figure 4.1: Naive implementation of a 1D Jacobi stencil.

Perhaps surprisingly, such simple algorithms are challenging to implement efficiently. Not only are they often applied to very large domains (requiring a large amount of computing power to meet performance needs), but each update operation also typically involves many earlier results (up-to 29 in some applications). For these reasons, naive implementations are severely memory-bound, since values need to be fetched redundantly from external memory.

The main goal of this chapter is to expose these obstacles and strategies to overcome them. It is organized as follows. In Section 4.2, we give some basic definitions on stencil computations. In Section 4.3, we motivate the relevance of these concepts with selected examples. In Section 4.4, we discuss the main implementation challenges in the formalism of the roofline model [33]. In Section 4.5, we introduce

the tiling transformation, a fundamental tool in the implementation of regular algorithms such as stencils. In Section 4.6, other forms of tiling are presented, that present varying advantages in terms of parallelism or external communication. In Section 4.7, we discuss the problem of memory allocation. We conclude in Section 4.9.

4.2 Definitions

A d-dimensional stencil is an iterative computation over a d-dimensional grid (or array) of data. The grid is iteratively updated a problem- or instance-dependent number of times, called *timesteps*. At each timestep, the value of every point in the grid is recomputed from the value of its neighbors, according to a *uniform* (fixed) dependence pattern. Precisely, let $A(t, \vec{x})$ denote the value of the grid at point $\vec{x}$ and timestep t. It is defined by a relation of the form:

$$A(t, \overrightarrow{x}) = f\left(\overrightarrow{x}, A\left((t, \overrightarrow{x}) + \overrightarrow{d_0}\right), \ \ldots, A\left((t, \overrightarrow{x}) + \overrightarrow{d_{m-1}}\right)\right), \tag{4.1}$$

where m is the number of dependences and the $\overrightarrow{d_i}$'s are constant *dependence vectors*.

4.2.1 Classification

Spatial invariance: Remark that, in the general case, the computation may depend on spatial position $\overrightarrow{x}$. When such is not the case, except perhaps at domain boundaries, we call the stencil *spatially-invariant*. Spatially varying stencils typically depend on position-specific coefficients. However, in our definition, the update formula does not depend on t and the computation for any given point is thus the same acrosse time iterations.

Access patterns: Stencils usually classified based on their dependence patterns:

- A stencil with **Jacobi**-style dependences (or *Jacobi stencil*) is a stencil where new values are computed exclusively from previous timesteps. Dependence vectors are of form: $(-k, \overrightarrow{v})$ with $k > 0$.
- Otherwise, it is called **Gauss-Seidel**. Dependence vectors are of form $(-k, \overrightarrow{v})$ with $k \geq 0$, with $k = 0$ for at least one dependence.

The names *Jacobi* and *Gauss-Seidel* refer to eponymous iterative methods for solving systems of linear equations. These methods are themselves stencil computations, with *Gauss-Seidel* slightly improving upon *Jacobi* by using more recent results to improve convergence.

4.2.2 Boundary Conditions

The equation 4.1 only properly defines the computation in the interior of the grid ; boundary points, where dependence vectors reach outside of the domain, need special handling. Many forms of boundary conditions have been proposed, depending on application requirements. For example:

- Homogeneous Dirichlet boundary conditions, where values are set to 0 at grid boundaries, are often used in image processing applications.
- In physics simulations, more exotic schemes such as *reflecting*, *absorbing* or *periodic* boundary conditions are often used. Reflecting and periodic boundary conditions ensure that the energy of the system stays constant over time, while absorbing conditions may be used to simulate an infinite field by letting energy dissipate outside of the grid.

In this chapter, we mostly ignore boundary conditions, as they typically account fo a small fraction of the computation and do not represent a major performance factor.

4.3 Examples

In this Section, we illustrate the above definitions with various examples from different fields.

4.3.1 Cellular Automata

Cellular (finite-state) automata (CA), such as Conway's *Game of Life* (GoL), [34] are famous examples of stencil computations. In GoL, points (called cells) admit only two states: *dead* or *alive*. Between consecutive timesteps, live cells switch off if they have less than 2 or more than 3 extant neighbors, while dead cells turn to life if they are surrounded by exactly 3 live cells. As an example, Figure 4.2 illustrates a common GoL pattern known as the glider.

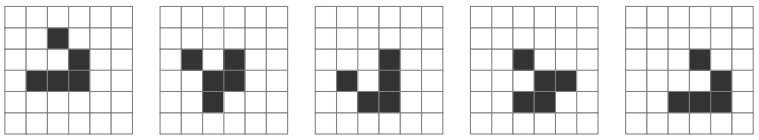


Figure 4.2: Glider pattern in Game of Life.

CAs have many practical applications – they are used, for example, in the simulation of physical [35] and biological systems [36]. They are also studied from a theoretical point of view for their ability to exhibit complex global behavior out of local rules and configurations [37].

However, automata are uncommon examples of stencils in that cells can only take a small number of values. Thanks to this small state-space, GoL and related automata lend themselves very well to implementation techniques based on memoization, such as the *HashLife* algorithm. In this thesis, we mostly focus on numerical stencils, which expose a virtually infinite state-space. For such stencils, memoization is impractical and the computation rules must be applied repeatedly for each point in the grid.

4.3.2 Smith-Waterman

Smith-Waterman is a dynamic programming algorithm for sequence alignment, with applications in bioinformatics. A sequence is defined as a list of *symbols* from a finite alphabet Σ (for example, a protein is a sequence of nucleotids). The Smith-Waterman algorithm determines the similarity between two biological sequences $A = a_1 \dots a_m$ and $B = b_i \dots b_n$ by inserting "gaps" within A and B to find the best alignment (in a given sense). Those gaps, called *indels*, model the insertion or deletion of a symbol in either A and B since the sequences diverged from a hypothetical common ancestor. The higher their similarity, the more likely they are to be historically related.

For example, let $\Sigma = \{A, B, C\}$. Consider the following alignments of sequences:

```
BBAABAC          BBAABAC--
ABA-BAD          ---ABABAD
```

Both are valid, as the resulting strings have the same length. However, one clearly represents a more plausible biological evolution, as it maximizes the number of matching symbols and reduces the count of indels.

To model this intuition, we assign a similarity score $D(a, b)$ to any pair of symbols, and affect a penalty W to the introduction of an indel in either A or B. The best alignment score for the two sequences is built iteratively from that of their prefixes, by filling out a matrix H of size $m \times n$ with the following formula:

$$H(i,j) = \max \begin{cases} 0 \\ H(i-1, j-1) + D(a_i, b_j) \\ H(i-1, j) - W \\ H(i, j-1) - W \end{cases}$$

The best score is given by $H(m, n)$. The alignment itself can be retrieved by walking backwards to reconstruct the series of insertions/deletions and substitutions.

The computation of the score matrix is a 1D stencil, where each row corresponds to a timestep. Since $H(i, j)$ depends on $H(i, j-1)$, it has Gauss-Seidel dependences, with the following dependence vectors:

$$(-1, -1),\ (-1, 0),\ (0, -1).$$

4.3.3 Finite-Difference Methods

The bulk of stencil computations arise from forward-time, *Finite-Difference* discretization of *partial differential equations* (PDEs). Depending on application, the output of such stencils is the state of the system after some predetermined simulation time, or when some equilibrium has been reached.

Stability of such numerical schemes usually requires smaller timesteps compared to backward, implicit methods, but are also simpler to implement. They typically result in large computational workloads, especially in High-Performance Computing world.

Heat Equation The heat equation is one of the canonical examples of stencils. It models the transfer of heat in a medium over time. Consider a surface with uniform thermal conductivity. The function $u(t, x, y)$ giving the temperature at time t and position (x, y) is a solution to the following PDE:

$$\frac{\partial u(t,x,y)}{\partial t} = \alpha \left(\frac{\partial^2 u(t,x,y)}{\partial x^2} + \frac{\partial^2 u(t,x,y)}{\partial y^2} \right).$$

This equation can be discretized to the following stencil computation[1]:

$$\begin{aligned} U(t,x,y) = U(t-1,x,y) \\ &+ c_0 \left(U(t-1,x-1,y) + U(t-1,x+1,y) - 2U(t-1,x,y) \right) \\ &+ c_1 \left(U(t-1,x,y-1) + U(t-1,x,y+1) - 2U(t-1,x,y) \right), \end{aligned}$$

where $c_0 = \alpha \Delta t / \Delta x^2$ and $c_1 = \alpha \Delta t / \Delta y^2$ are constant coefficients depending solely on the discretization steps along each dimension. It thus defines a 5-point, spatially-invariant Jacobi stencil with the following dependence vectors:

$$(-1,0,0),\ (-1,-1,0),\ (-1,1,0),\ (-1,0,-1),\ (-1,0,1)$$

Boundaries may be handled, for example, by fixing $u(t, x, y) = K$ outside the surface.

In general, the coefficients c_0 and c_1 may depend on position (x, y), as thermal conductivity is not necessarily uniform. In such case, the stencil becomes:

$$\begin{aligned} U(t,x,y) = U(t-1,x,y) \\ &+ c_0(x,y) \left(U(t-1,x-1,y) + U(t-1,x+1,y) - 2U(t-1,x,y) \right) \\ &+ c_1(x,y) \left(U(t-1,x,y-1) + U(t-1,x,y+1) - 2U(t-1,x,y) \right) \end{aligned}$$

and hence loses the property of space-invariance.

Note that we may recover this property by switching to a *multi-field* stencil. Indeed, the definition of stencils does not constrain update operations to a compute a single-value ; way me choose to embed field coefficients with actual values, simply propagating coefficients from one time iteration to the next.

Seismic Modeling Finite-difference stencils are also used in seismology, for example to model the propagation of seismic waves in a medium. This problem is called *seismic modeling*. Specifically, one is interested in computing the pressure field P after a pre-determined simulation time, given some initial conditions and a space-varying *propagation velocity* field v.

As with most PDEs, many finite difference schemes may be used to discretize this problem, resulting in stencils of varying numerical complexity and stability properties. A simple 6-point scheme, spanning *two* iteration steps, is illustrated in Figure 4.3. The update equation is of form:

$$\begin{aligned} P_{x,y,t} = b_{x,y} + P_{x,y,t-1} - P_{x,y,t-2} + \\ a_{x,y} \left(P_{x+1,y,t-1} + P_{x-1,y,t-1} + P_{x,y+1,t-1} + P_{x,y-1,t-1} \right) \end{aligned}$$

[1]This is a Forward-Time, Central-Space (FTCS) discretization. Other discretization schemes may be used, resulting in stencils with different numerical properties in terms of stability and accuracy.

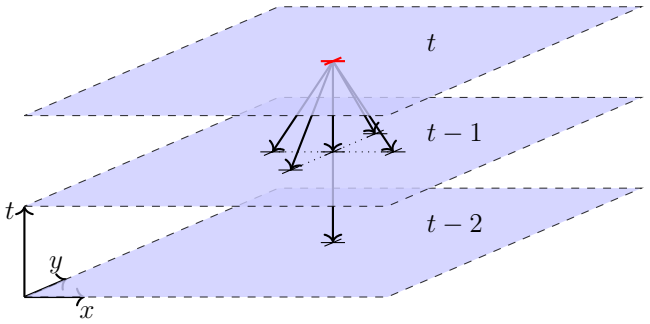


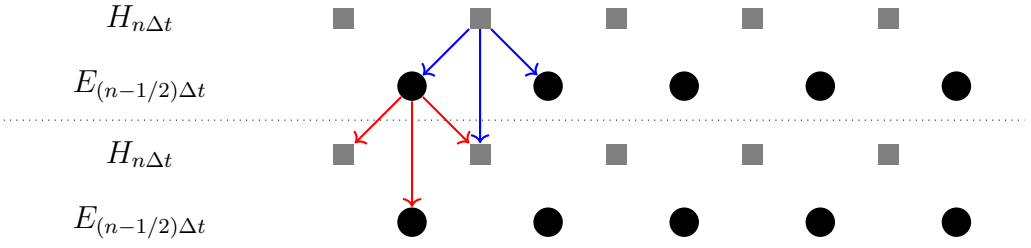
Figure 4.3: Seismic Modeling.

Fields a and b represent spatially-varying coefficients, computed from the derivatives of the velocity field. The same remarks as for the heat equation regarding spatial invariance also apply here.

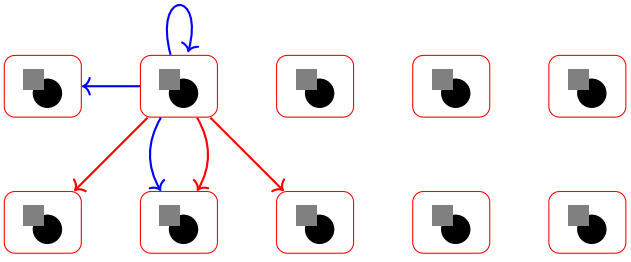
Yee's Algorithm Yee's algorithm for Maxwell equations is one of the first finite-difference methods for PDEs. It simultaneously solves for the electrical and magnetic fields E and H in the time domain.

At first glance, this method does not exactly match the definition of stencils given at the beginning of this Section. Let Δt be the size of time iterations. At the n-th timestep, fields E and H are mutually recomputed at time $(n - 1/2)\Delta t$ and $n\Delta t$. Moreover, in cartesian 3D space, the discretization lattices used for E and H are offset from one another by $(\Delta x/2, \Delta y/2, \Delta z/2)$ so that each point of E (or H) is surrounded by 6 points of H (or E) along the canonical axes.

To simplify the discussion, we will restrict ourself to the 1D case. The spatial and temporal decomposition of the iteration space is illustrated by the picture below, along with the data dependence vectors:



We can transform this scheme to a stencil in the conventional sense by "grouping" elements of E and H diagonally:



In this form, Yee's algorithm is a multi-field Gauss-Seidel stencil, where each point computes two values out of four other groups. The dependence vectors are:

$$(-1,-1),\ (-1,0),\ (-1,1),\ (0,-1)$$

The apparent self-dependency in the above diagram only imposes an order in the computation of E and H within the same iteration.

4.4 Implementation Challenges

Performance of scientific kernels (such as stencils) is the result of complex interplay between hardware resources (cache/local memory, bandwidth, computing power), architectural behavior and application-/implementation-specific factors.

In the case of stencils, the most important performance factor is the balance between computations and communication. An implementation is said *compute-bound* if its performance is ultimately limited by the available amount of computing power ; in contrast, it is said *IO-bound* if computing power is in excess compared to the available memory bandwidth. The balance between both is the prominent challenge in the implementation of stencils.

This problematic may be conveniently exposed in the formalism of the roofline model [33], an intuitive tool for understanding performance characteristics of parallel implementations. It relates performance of numerical algorithms, measured in *GFlops*[2], *i.e.*, with their *arithmetic intensity* and the characteristics of the implementation platform.

Arithmetic intensity I (also called *compute/IO ratio*) is the implementation-specific ratio, expressed in *Flops/B*, of computations over communication volume. Let β be the bandwidth limit (in GB/s) of our architecture. We see easily that the maximum performance, achievable via parallelization (not taking other resource constraints into account) is simply:

$$10^{-6} \times \beta \times I \quad \text{(GFlops)}.$$

Since β is a constant, the only way to improve this throughput is thus to reduce memory accesses to increase arithmetic intensity .

Naturally, performance is also ultimately limited by the peak throughput P of the architecture (for example, on a multi-core architecture, throughput is limited by the number of cores and their frequency). We may thus derive a bound on achievable performance:

$$\min(P,\ 10^{-6} \times \beta \times I) \quad \text{(GFlops)}.$$

All these phenomena are illustrated in Figure 4.4. Arithmetic intensity (in flops/byte) is represented in the horizontal axis, while attainable performance (in GFlops) is displayed in the vertical axis. This plot only gives coarse-grained, conservative performance estimates; in most cases, other factors such as bad data locality

[2]In the case of stencils, the number of updates per second may be a more meaningful metric ; both are related by a constant, application-dependent factor (the number of operations per update) and are thus equivalent.

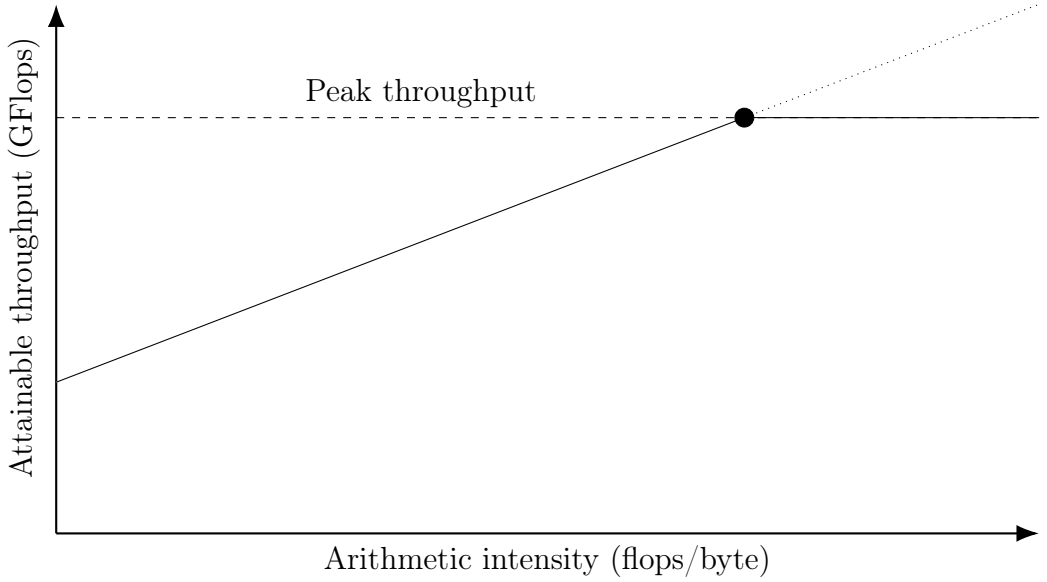


Figure 4.4: Roofline Model

also negatively affect the attainable throughput. This model may be refined to account for other limitations, materialized as additional ceilings; for example, peak throughput may be lower without SIMD support. Actual bandwidth limits also depend on a variety of factors, such as access patterns, which can degrade attainable throughput by introducing additional latency.

Finally, observe that arithmetic intensity is ultimately capped by the amount of local memory ; indeed, it can only be improved by reducing the number of memory accesses via buffering. The tiling transformation, exposed in the next sections, gives a mean to control tradeoffs between computation, communication and memory usage.

4.5 Tiling Transformation

Tiling is a fundamental tool in many implementations of stencil computations. Its first use is to improve data-locality, by partitioning the computation into smaller chunks, called tiles, that can fit on local memory. Its second purpose is to extract coarse-grained parallelism, to dispatch batches of work to multiple processing elements, or pipeline the execution of independent tiles on the same datapath. For example, most GPU stencil implementations map tiles to different thread blocks, where fine-grained parallelism is used to concurrently execute independent computations on multiple threads [38]. Intermediary values can thus be kept in block-local shared memory, without external memory accesses.

Multiple flavors of tiling have been proposed in the literature. In this Section, we introduce the fundamental ideas of tiling with *rectilinear* (or *hyper-parallelepipedic*) tiling. Like most tiling methods, it consists in partitioning the iteration space into convex regions, according to carefully chosen *tiling hyperplanes*. We will see in Section 4.6 that different hyperplanes may be used to expose trade-offs in terms of

```
float A[T+1][N+1] = ...;

for (int t=1; t<=T; t++)                    // Temporal loop
  for (int x=1; x<=N; x++)                  // Spatial loop
S:  A[t][x] = f(A[t-1][x], A[t][x-1]);
```

Figure 4.5: Naive Gauss-Seidel stencil implementation

parallelism and communication behavior. However, most of the terminology remains the same and is better understood in a simple context.

4.5.1 Iteration Space and Dependences

Consider the affine stencil loop-nest in Figure 4.5. As you may recall from Chapter 3, we can represent all instances of statement `S` as a (convex) polyhedral domain:

$$\mathcal{D}_{\mathtt{S}} = \{(t, x) \in \mathbf{Z}^2 \mid 1 \leq t \leq T \wedge 1 \leq x \leq N\}.$$

$\mathcal{D}_{\mathtt{S}}$ is called the *iteration space* of statement `S`. As any polyhedron, it is the intersection of finitely many half-spaces, determined by affine constraints. In this case, affine constraints are simply derived from loop bounds, which depend on parameters T and N.

Most iterative stencils may be implemented as such perfectly-nested loops, where the outermost loop iterates over timesteps, and the *d-th* innermost loops scan the spatial grid. A d-dimensional stencil hence gives rise to a $n = (d + 1)$-dimensional iteration domain.

The iteration space of statement `S` can be conveniently represented as a regular lattice of integral points (see Figure 4.6). Each point represents an *instance* (or *execution*) of statement `S`. Its coordinates are the values of of iterators (t, x).

4.5.2 Schedule

In Figure 4.6, the *iteration order* of the original program is represented as dashed path. Since the loops iterate over each dimension in increasing order, it simply corresponds to a lexicographic scan of the domain:

$$(1, 1), (1, 2), \ldots, (1, N), (2, 1), (2, 2) \ldots, (2, N), \ldots, (T, 1), (T, 2), \ldots, (T, N).$$

In general, iteration order may be linked to that of a schedule. Given an iteration domain $\mathcal{D}_S$, a schedule is a map $\Theta : \mathcal{D}_S \to O$ where O is a set equipped with a partial or total order $\preceq$. The relative execution order between two instances $\vec{i}, \vec{j} \in \mathcal{D}_S$ is given by that of their image in $(O, \preceq)$ by Θ. Precisely:

$$\vec{i} \prec \vec{j} \Rightarrow \vec{i} \text{ executed before } \vec{j}.$$

An interesting class of schedules is given by *affine* schedules, the set of piecewise, quasi-affine maps to some lexicographically ordered polyhedron. For example, in the

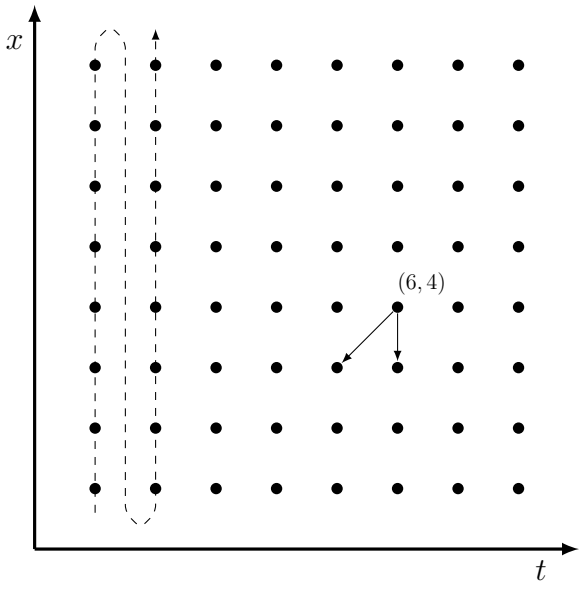


Figure 4.6

initial stencil, $O = \mathcal{D}_S$ and the schedule is simply the *identity* map. The number of dimensions in polyhedron O is called the dimensionality of the schedule.

Let $I = \Theta(\mathcal{D}_S) \subset \mathbf{Z}^m$ stand for the image of domain $\mathcal{D}$ by an affine schedule Θ. For simplicity, let us assume that Θ is an injective function, i.e., each point in I has a single pre-image in $\mathcal{D}$. When such is the case, the schedule defines a total execution order. In pseudo-code, it corresponds the following program:

$$\textbf{lexfor}(\vec{i} \in I) \qquad \mathtt{S}[\Theta^{-1}(\vec{i})];$$

where **lexfor** denotes a lexicographic domain scan.

In other words, the program scans the schedule *image* of the domain in lexicographic order, and executes the corresponding pre-image instance. Under some conditions, it is possible to modify the execution order of the program without altering its semantics ; in the formalism of the polyhedral model, we often apply combinations of affine transformations to the co-domain of the schedule, before regenerating imperative code. However, the problem of turning the **lexfor** construct above into efficient code is a non-trivial problem that was not truly solved until 1991 by Ancourt et al. [39]. More recently, Boulet et al. [40] proposed to implement this scan as a state machine, by computing the *next* relation, mapping each iteration point to its successor in the domain.

4.5.3 Dependence Relation

The schedule in the original program is simply:

$$\{\mathtt{S}[t, x] \rightarrow [t, x]\},$$

Since each statement writes in a different memory cell, we easily see that data accesses induce the following dependence relation:

$$\{\mathtt{S}[t,x] \rightarrow \mathtt{S}[t',x'] : t' = t-1 \vee x' = x-1\},$$

In more complex cases, dependence analysis techniques such as Array Dataflow Analysis [26], discussed in the previous chapter (see Section 3.4.3), may be used to determine the dependence relation of polyhedra programs, such as stencils, automatically. Intuitively, this relation defines a validity condition for any schedule: an instance must always be executed after all its dependences.

Because of our single-assignment addressing, our program only has *true* or *data*-dependences: a statement instance depends on another *iff* it uses the value produced by that instance. In many cases, to reduce memory usage, some form of *memory contraction* must be implemented (see Section 4.7). For example, the result of instance $\mathtt{S}(t,x)$ could be mapped to `A[t%2][x]` without changing semantics under the initial schedule. However, such redundant memory allocations may result in *false* or *spurious* dependences, which may impede re-scheduling or prevent parallelization. However, for polyhedral programs, such dependences can always be eliminated [Feautrier SSA], in order to separate scheduling and memory allocation concerns.

Consider an arbitrary schedule $\Theta : \mathcal{D}_S \rightarrow \mathbf{Z}^m$. We say that Θ is valid if it is compatible with the dependence relation defined by the original program, in the following sense:

$$\forall (t,x),(t',x') \in \mathcal{D}_S, \quad (t',x') \rightarrow (t,x) \Rightarrow (t,x) \prec (t',x').$$

As we sill see, tiling consists in defining a new, higher-dimensional schedule for the original stencil.

4.5.4 Tiling

In the original schedule, consider the number of steps between the production of a value and its first reuse by a later iteration. The value produced by instance (t,x) is first re-used by iteration $(t+1,x)$: the reuse distance is thus equal to N, the spatial size of the domain. If N is large enough, the value will be evicted from cache before being next accessed. Ignoring domain boundaries, cache miss rates are thus of 50%. Since RAM accesses can take hundreds of CPU cycles on modern architecture, this problem can significantly affect performance. It is even more severe for higher-dimensional stencils, as in numerical solvers, the size of the grid typically grows exponentially with dimensionality.

Tiling addresses this issue by partitioning the domain into tiles arbitrary size, in order to improve locality behavior. Figure 4.7 illustrates rectangular tiling for our running example. A fundamental idea of tiling is that tiles must be *atomic*, i.e., there must exist a valid schedule of instance where the execution of distinct tiles do not overlap. An example of such a schedule, corresponding to a lexicgraphic ordering of tiles, is represented in Figure 4.7 as a dashed path. As we will see, this schedule is not unique – others can be used, for example, to harvest inter-tile parallelism.

The pattern in the figure corresponds to a 4-dimensional schedule: the outermost two dimensions correspond to the space of tiles, while innermost dimensions

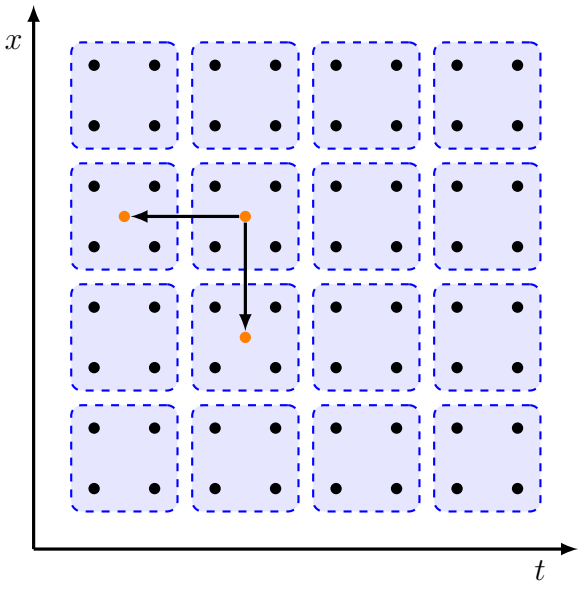


Figure 4.7: Tiling of a 1D Gauss-Seidel stencil with 2×2 rectangular tiles.

correspond to intra-tile iterations. The schedule may be described by the following quasi-affine map:

$$\left\{ \mathtt{S}[t, x] \to \left[\left\lfloor \frac{t-1}{2} \right\rfloor, \left\lfloor \frac{x-1}{2} \right\rfloor, (t-1) \bmod 2, (x-1) \bmod 2 \right] \right\}$$

Asuming that 2, the size of tiles, divides T and N evenly, this tiling may be implemented in source code as in Figure 4.8.

Tiling improves locality by reducing the re-use distance between two intra-tile iterations, which is now bounded by the volume of the tile. Tile size thus gives a mean to control temporal locality, enhancing performance under memory and bandwidth constraints, since results can now be kept in fast memory, reducing the need for external memory accesses. Thanks to this property, tiles are often used as communication boundaries: results produced within a tile are kept in cache available for later iterations, while outer dependencies must be fetched from elsewhere (e.g., loaded from shared memory sent by another node on the network).

4.5.5 Tile-Level Dependencies

Instance-level dependencies naturally induce *tile-level* dependencies. We say that a tile depends on another tile if a point in the first tile depends on a point in the second tile. Because of atomicity, we may view the result of tiling as a new stencil, where tiles take the role of statement instances.

In the Gauss-Seidel example, this new stencil happens to have the same dependences as the untiled stencil. However, this needs not be the case. For example, consider the tiled Jacobi stencil in Figure 4.9. Observe that instance-level dependencies cross the barrier between two tiles in both directions: in other words, with this tiling, the tile-level stencil has a circular dependence. There is naturally no

```
float A[T+1][N+1] = ...;

// Iterations over tiles
for (int t0=1; t0<=T; t0+=2)
  for (int x0=1; x0<=N; x0+=2)
/* ———————— External communication boundary ———————— */
    // Intra-tile iterations.
    for (int t1=0; t1<2; t1++)
      for (int x1=0; x1<2; x1++) {
        int t = t0+t1;
        int x = x0+x1;
        A[t][x] = f(A[t-1][x-1], A[t-1][x]);
      }
```

Figure 4.8: Tiled source code of the 1D Gauss-Seidel stencil.

schedule of instances that is atomic for tiles and compatible with this dependence relation, and this tiling is invalid.

4.5.6 Skewing

The problem of the stencil in Figure 4.9 is the bi-directionality of data flow across the hyperplane normal the horizontal dimension. In a more general sense, a tiling of an n-dimensional iteration space is defined by n *tiling hyperplanes* $(\phi_0, \ldots, \phi_{n-1})$, and size / offset parameters. It can be shown [41] that a tiling is valid if the projection dependence vectors along each normal to tiling hyperplanes is of the same sign: this condition captures the necessity of uni-directionality we already mentioned.

In rectilinear tiling, tiling hyperplanes are simply the canonical hyperplanes, normal to one of the canonical basis vector. For our Jacobi stencil, dependence vectors are:

$$(-1,-1),\ (-1,0),\ (-1,1)$$

The vertical tiling hyperplane is normal to the $(0,1)$ basis vector. We see that:

$$(-1,-1)\cdot(0,1) = -1 \quad \text{whereas} \quad (-1,1)\cdot(0,1) = 1,$$

which are of opposite signs. One way to address this issue is to apply rectilinear tiling to a skewed iteration space, where unidirectionality with respect to canonical hyperplanes has been enforced.

For example, consider the domain transformation:

$$\{S[t,x] \to S'[t,x+t]\}.$$

We may apply this transformation to the dependence vectors to verify that rectilinear tiling is now valid:

$$(-1,-2),\ (-1,-1),\ (-1,0).$$

All dependence vectors now have non-positive components along each dimension: rectilinear linear tiling can now be applied.

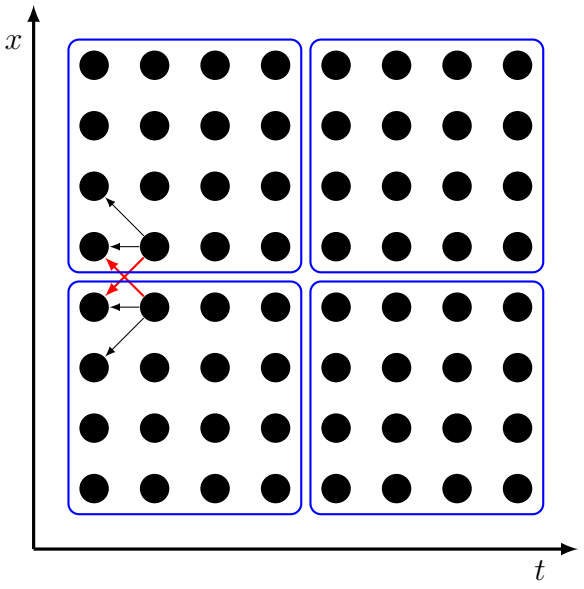


Figure 4.9: Invalid tiling for a Jacobi skewing.

The result of skewing+tiling for the Jacobi stencil is illustrated in Figure 4.10. Inevitably, tiling after skewing introduces incomplete tiles at domain boundaries. If the domain is small compared to tile size, partial tiles can account for a large proportion of tiles and introduce a significant overhead, as their implementation cannot be fully specialized.

It is to be noted that writing efficient implementations of skewed and tiled stencils by hand is a challenging task. When tile dimensions are compile-time constants, polyhedral code generation techniques can be applied. For example, the Pluto compiler features efficient algorithms for finding valid tiling hyperplanes and performing source-to-source tiling transformations. However, when tile dimensions are dynamic parameters, tiling can no longer be seen as a polyhedral transformation. Parametric code generators have been proposed, for example, for rectilinear tiling [42–44]. Such generators can allow dynamic tile-size tuning.

4.5.7 Tile Halo and Communication Volume

The set of external dependencies of a tile is called its *halo*. For 1D stencils, halo volume grows linearly with tile size, while computation volume grows quadratically; this fact generalizes to higher-dimensional volumes as well. Since the halo of a tile can be seen as its (input) communication volume, it means that increasing tile size also improves the *compute/IO* ratio, thus reducing bandwidth requirements, at the cost of increased memory usage.

For multidimensional rectilinear tiles, a tight upper-bound of communication volume, based on the concept of *depth*, can be given. Let $(d_j)_{1\leq j<M}$ be the dependence vectors of the stencil. For reasons that will be made clearer, we require that dependence vectors all have non-positive coordinates. Dependence depth along the

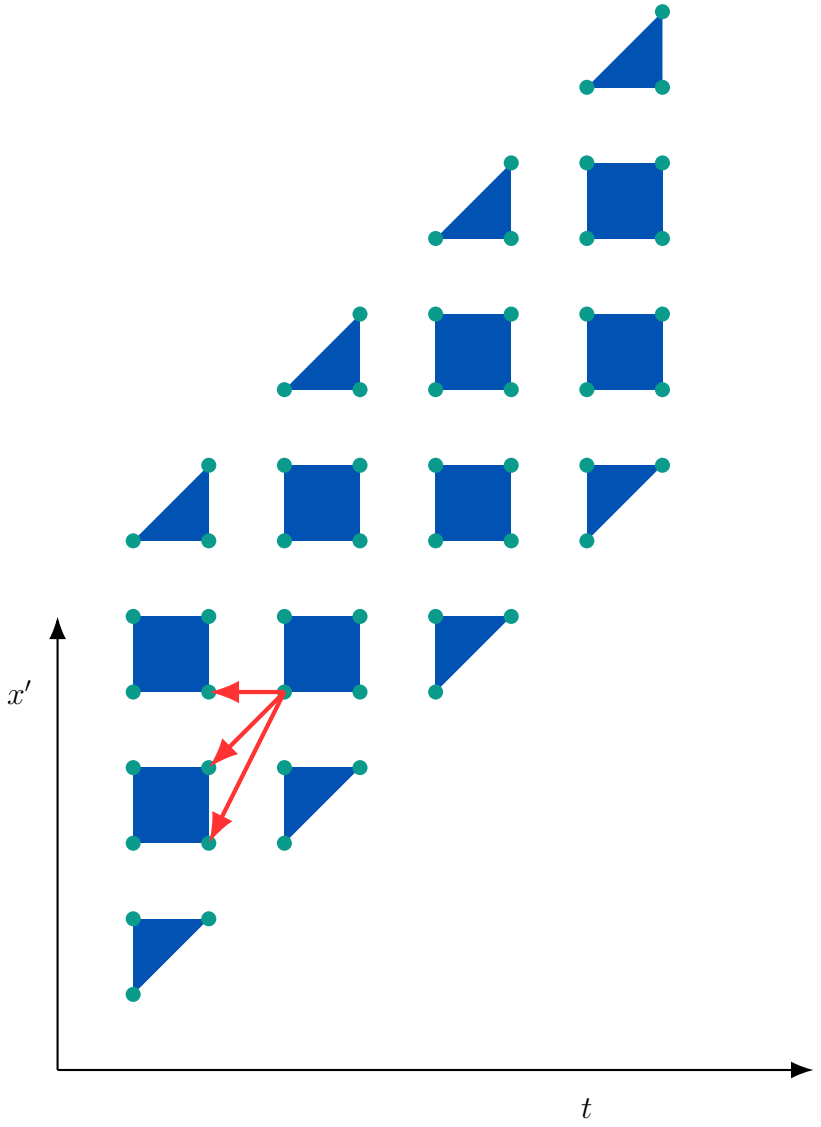


Figure 4.10: Legal tiling for the Jacobi 1D stencil after skewing transformation.

n-th dimension is defined as:

$$\delta_n = \max_j(-d_j \cdot 1_n),$$

where 1_n denotes the 1_n-*th* canonical basis vector.

Let S_n denote tile size in the nth dimension. Then, communication volume is bounded by:

$$\sum_{i=0}^{d} \delta_i \prod_{j \neq i} S_j$$

Consider the restricted case where $S_i = S$ for all i. Then, this bound simplifies to:

$$(\sum_i \delta_i) S^d$$

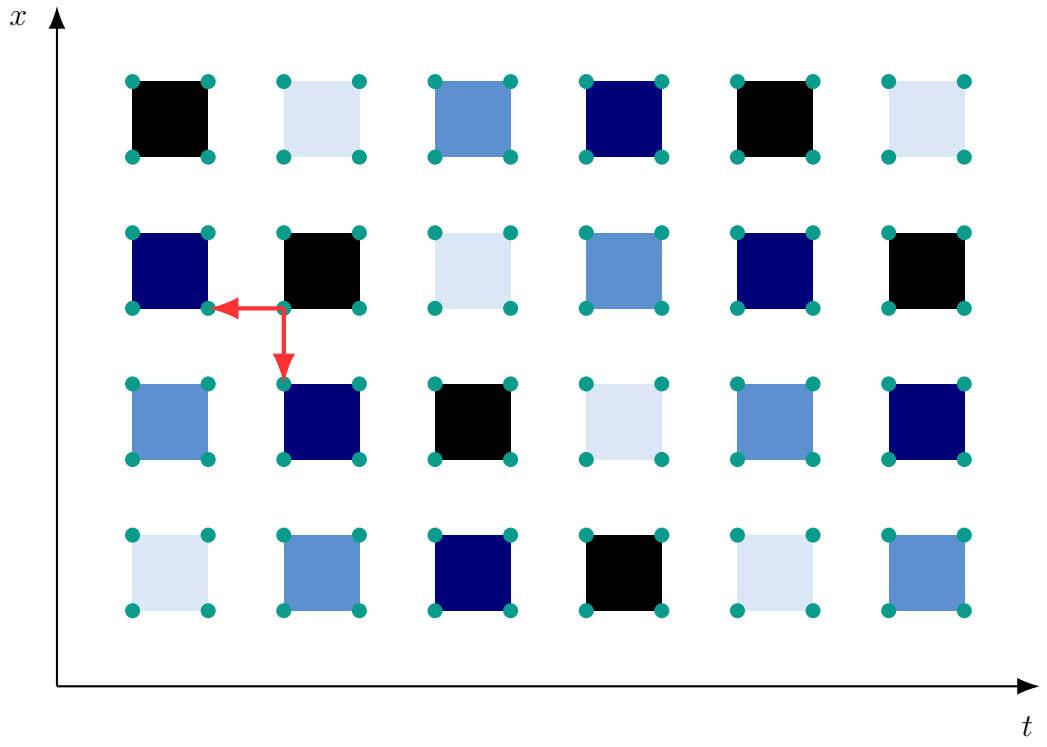


Figure 4.11: Tile wavefronts for the 1D Gauss-Seidel stencil.

While the volume of the tile is S^{d+1}. The ratio between computation and communicaton is thus:

$$\frac{S}{\sum_i \delta_i},$$

and thus increases linearly with size S.

4.5.8 Wavefront Parallelism

Wavefront parallelism has been initially introduced for instance-wise parallelism, and is not directly linked to tiling. A wavefront is a hyperplane of parallel instances in regular commputations such as stencils. When a stencil is tiled, one is interested in finding sets of independent tiles for parallel execution.

In Figure 4.11, tile wavefronts are pictured in alternating colors for the Gauss-Seidel stencil. Observe that, because wavefronts scan diagonals of the domain, first and last wavefronts contain less tiles, and thus, less parallelism than others. This problem is called *load imbalance*, and its elimination is one of the main motivation behind many non-rectilinear tiling techniques.

Extracting wavefront parallelism may once again be seen as an application of skewing. It consists in exposing parallelism in the outer $n-1$ schedule dimensions, through a suitable domain transformation. For example, we may apply the following skewing to the domain of tiles of the Gauss-Seidel stencil:

$$\{S[t,x] \to S[t+x,x]\}$$

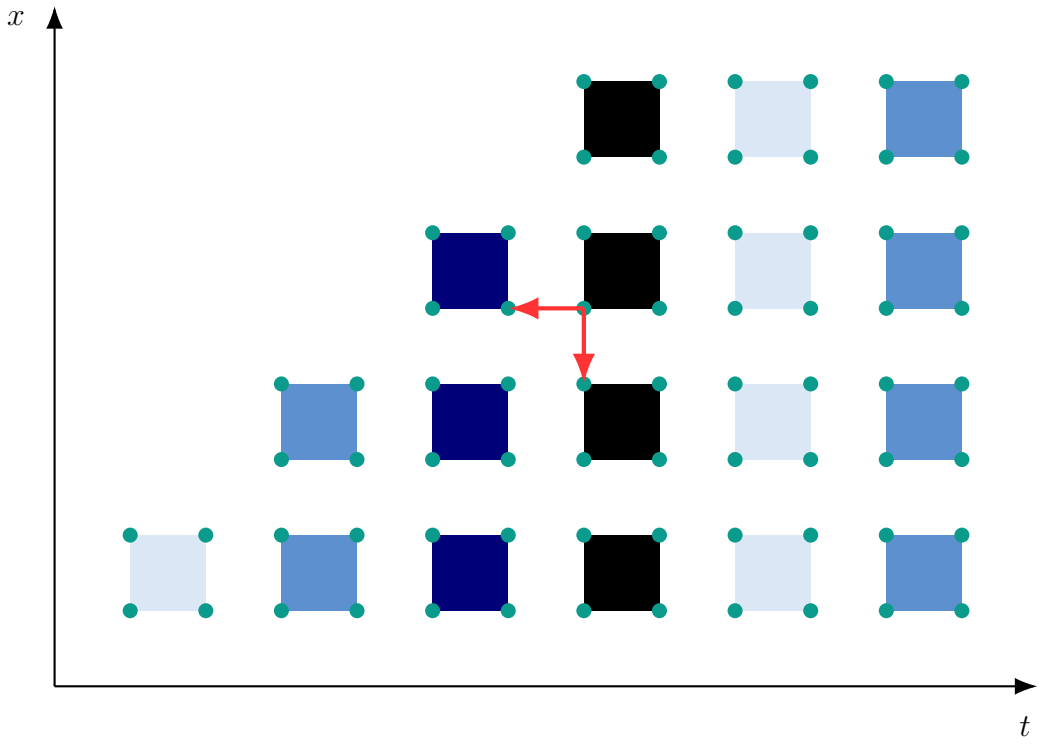


Figure 4.12: Wavefront parallelism extraction as an application of skewing (Gauss-Seidel stencil).

We can see in Figure 4.12 than wavefronts actually correspond to vertical bands of tiles in the transformed tile domain.

4.5.9 Hierarchical Tiling

In this Section we only considered one level of tiling. In order to accomodate multiple levels of memory hierarchy, a recursive tiling strategy may be adopted, by splitting tiles repeatedly. For example, smallest tiles may be unrolled for register-level tiling, while largest ones serve as coarse-grained communication units.

While in most hierarchical approaches, the same tiling hyperplanes are used atl all levels, hierarchical tiling can be used to expose nested parallelism through further domain transformations. For example, jagged tiling allows better control of the grain of intra-tile parallelism [45].

4.6 Tiling Variants

Rectilinear tiling offers a convenient way to extract coarse-grained parallelism from regular computations. However, it suffers from a significant drawback called *load imbalance*. Indeed, in hyper-parallelepipedic domains, tile wavefronts usually contain varying numbers of tiles: at the beginning and the end of the computation, there is thus not enough parallelism to evenly dispatch tiles on multiple processors. As a synchronization point is required after each wavefront to enforce inter-tile

dependences, this results in a suboptimal use of computational resources, as some processing elements need to wait for others to finish their work. This problem is especially severe on small grid sizes, since the number of large wavefronts does not compensate for this waste of computing power. Dynamic tile scheduling may help mitigate the issue [46], by allowing to run some tiles earlier, but does not address the root of the problem.

In technical terms, for stencil computations, rectilinear tiling is said to lack the *concurrent start-up* property. We say that a tiling technique enables concurrent start along some (usually canonical) dimension i if tiles along the corresponding unit vector can be computed in parallel. This definition can be extended to any subspace of $\mathbf{Z}^n$. Concurrent start over several dimensions can be beneficial in two ways:

- First, it provides more coarse-grained parallelism.
- Secondly, it reduces the need for synchronizations.

In this Section, we review several techniques which all feature concurrent start along at least one canonical dimension. Each of them also exposes subtle trade-offs, *e.g.*, in terms of communication volume and fine-grained-parallelism.

4.6.1 Overlapped Tiling

The first technique we review is called *overlapped tiling* [47]. While most tiling transformations are *tesselating*, i.e., partition the iteration space into non-verlapping tiles, overlapped tiling enables concurrent start through redundant computations (see Figure 4.13). This method eliminates the need for communication along one or several dimensions, but comes with a significant computational overhead, especially for "tall" tiles spanning a large number of timesteps. Overlapped tiling is hence an interesting strategy on architecture where memory bandwidth is commonly the performance bottleneck, such as GPUs. Even then though, actual performance results from a complex interplay of factors, and tile shape selection is an important concern. Indeed, the amount of redundant computation increases with tile height, and can thus quickly become detrimental to performance. Moreover, as illustrated by Figure 4.14, the minimal tile and its halo may have a relatively complex ; to simplify the control flow, more data than necessary must often be transferred. The proportion of useful communication thus also decreases with tile height. Finally, the amount of "live" data within decreases at each time iteration, resulting in suboptimal use of accelerator memory resources.

4.6.2 Diamond Tiling

For Jacobi stencils, rectilinear tiling constrains one of the tiling hyperplane to be parallel to original spatial dimensions. Diamond tiling [48] lifts that restriction and determines tiling hyperplanes from the envelope of the cone spanned by dependence vectors (see Figure 4.15). The main benefit of this technique is to enable concurrent start along one spatial dimension. However, in most cases, tiles cannot be started

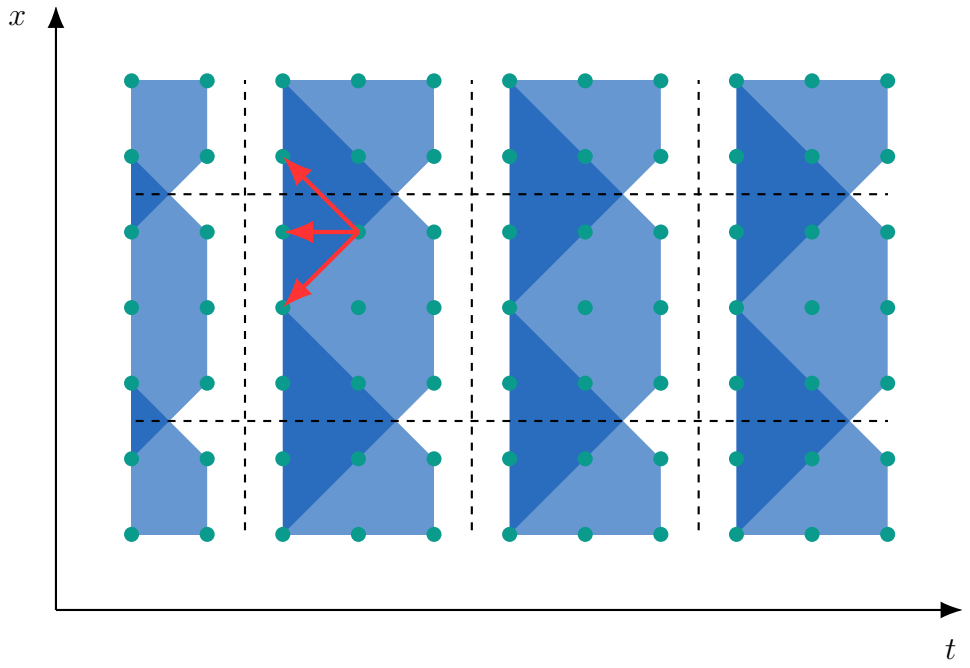


Figure 4.13: Overlapped tiling for a 1D 3-point Jacobi. Rectangular tiles of size 3×3 are extended to recompute all their dependencies down to the lower time boundary. Darker regions correspond to redundant computations.

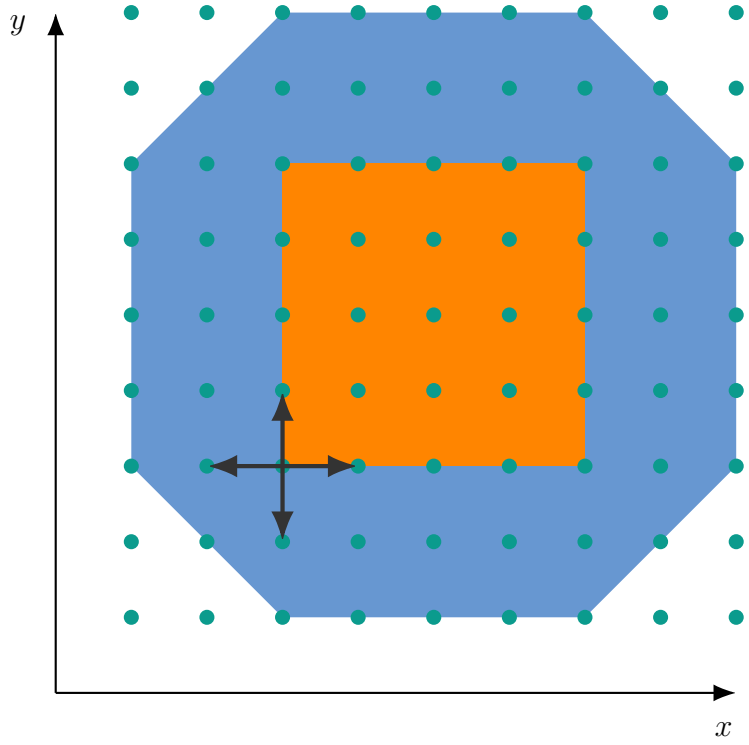


Figure 4.14: Shape of the input induced by a 2D Jacobi stencil for overlapped tiling, with tile size $2 \times 5 \times 5$. The shape of the top of the tile is displayed in orange.

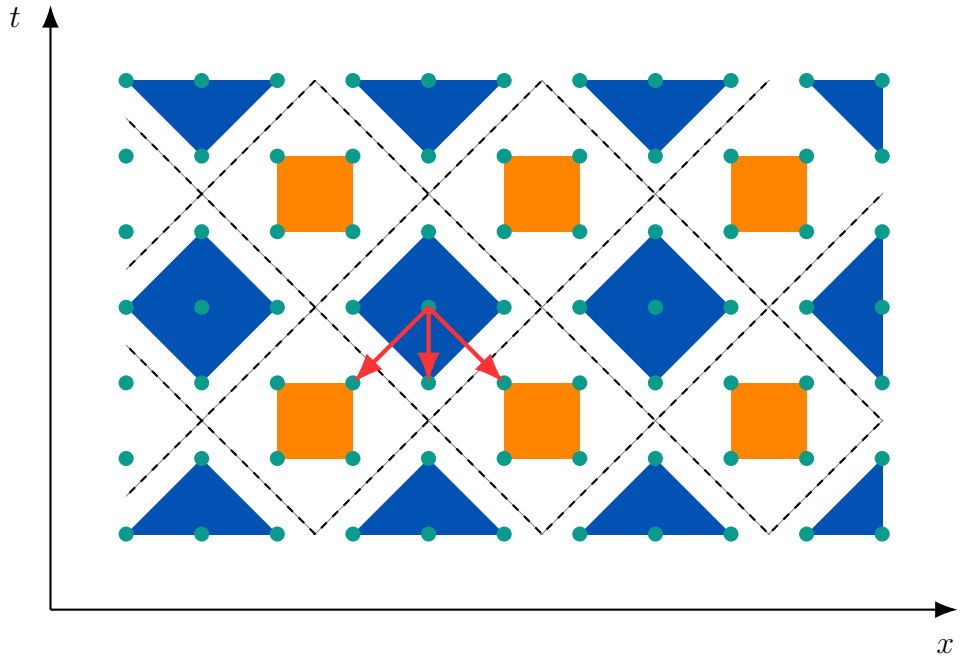


Figure 4.15: Diamond tiling for 1D 3-point Jacobi stencil. Tiling hyperplanes (dashed lines) are inferred from faces of the dependence cone, spanned by dependence vectors (red arrows). Each horizontal band of tiles can benefit from concurrent start. Note the heterogeneous shape of tiles in consecutive bands.

along all dimensions concurrently. Diamond tiling has proven efficient over conventional tiling on multi-core CPUs [49] and GPU architectures [50]. However, it suffers from a double-drawback: *i*) bandwidth requirements are irregular, since the first and last intra-tile timesteps require more memory accesses per iteration. This can lead to a computation being memory-bound, in spite of its *average* bandwidth usage not exceeding the architectural limit. *ii*) Similarly, first and last timesteps expose less parallelism than others, since tiles exhibit narrow "peaks" with fewer iterations. Consequently, intra-tile wavefronts exhibit the same pipelined startup pattern as tile wavefronts in conventional tiling. One may argue that diamond tiling does not truly solve the problem of load imbalance: it displaces it to a finer-grained level.

4.6.3 Hexagonal Tiling

Hexagonal tiling [51], shown in Figure 4.16 is a variant of diamond tiling that partially addresses the problems exposed above. Instead of deriving tile shapes from the narrowest dependence cone, tile peaks are "flattened" thus that each intra-tile timestep exposes a minimal level of parallelism. This strategy allows to reduce load and bandwidth imbalance. We observe that, as with diamond tiling, its full generalization to higher-dimensional stencils, while enabling concurrent start along all spatial dimensions, is not possible.

Another approach, developed for GPUs, is named *split tiling*. It combines some of the benefits and drawbacks of diamond and overlapped tiling. Like diamond

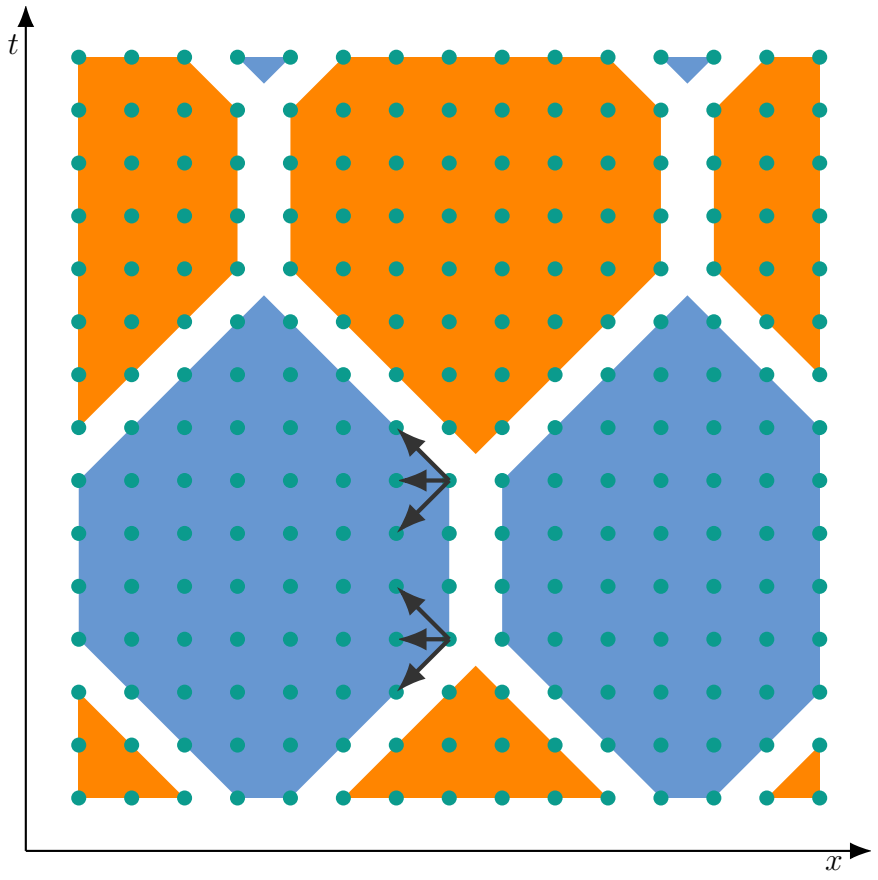


Figure 4.16: Illustration of hexagonal tiling, a generalization of diamond tiling. Unlike diamond tiling, the elongated top and bottom faces guarantee a minimum amount of intra-tile parallelism at each time step, while still allowing concurrent start along one of the iteration space boundaries.

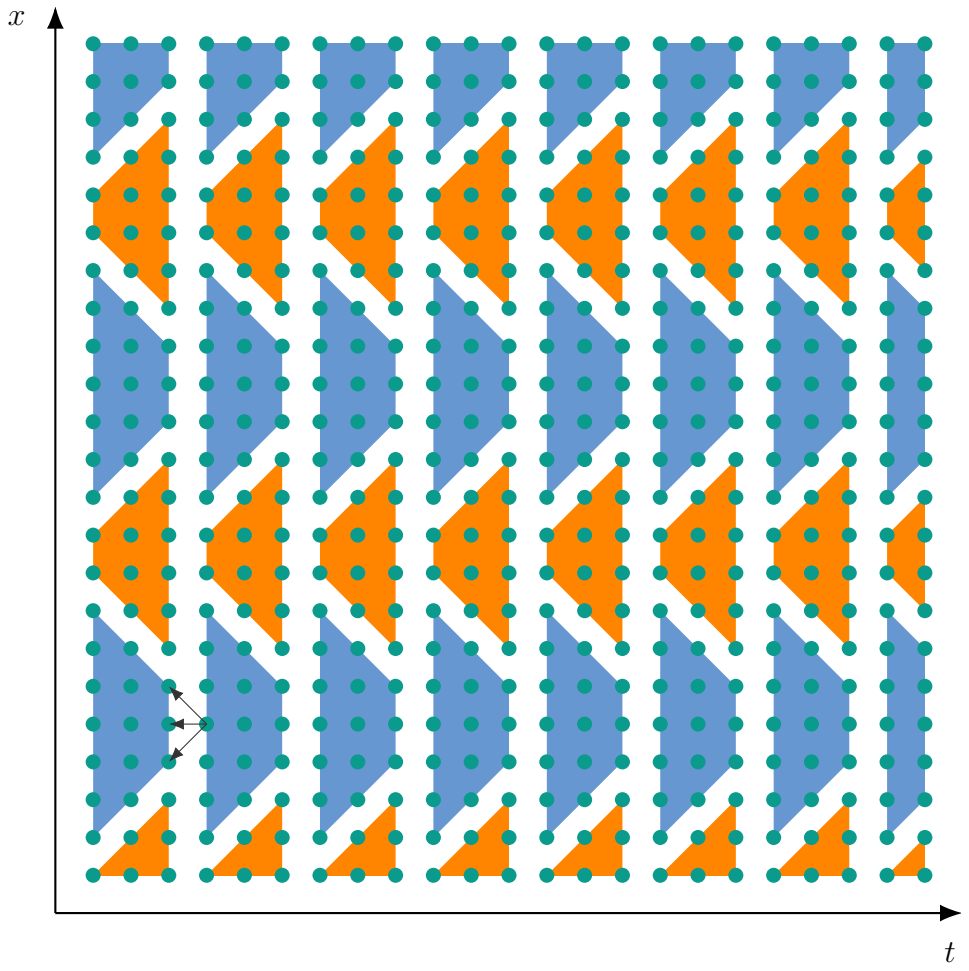


Figure 4.17: Similarly to hexagonal tiling, split tiling improves upon diamond tiling by enabling concurrent start while preserving fine-grained parallelism.

tiling, it enables concurrent start without introducing redundant computations. On the other hand, like overlapped tiling, it only requires external memory communication at time boundaries. The technique is illustrated in Figure 4.17. It consists in splitting each spatial wavefront in phases of heterogeneously-shaped trapezoidal tiles. These phases are executed sequentially, with output halos of the first phase kept in scratchpad memory for the second phase, reducing external bandwidth requirements. The main drawback of this approach is probably the large amount of shared memory needed to keep inter-tile dependencies on-chip and the complexity introduced by alternating tile shapes, which does not make this technique a good fit for architectures such as FPGAs, more optimized towards streaming access patterns with limited buffering.

4.7 Memory Allocation

Naive stencil implementations such as the one in Figure 4.5, where each memory cell is written at most once, are suboptimal in terms of memory usage. Indeed,

the memory footprint of the kernel increases linearly with simulation time, while the lifespan of each value does not exceed a few timesteps. As we will see, it is possible to reduce memory usage by re-using space affected to values that are no longer needed.

We call *memory mapping* or *memory allocation* a function assigning statement instances to memory addresses. For the purpose of this discussion, let us define mappings as functions of form:

$$\vec{i} \in S \mapsto \phi(\vec{i}) \in \mathbf{N},$$

where S denotes the iteration space of some statement S.[3] For example, if each statement $S(i,j)$ writes to address `S[i][j]`, where `S` is a multidimensional array of size $M \times N$, then memory allocation is given by:

$$\phi(i,j) = i \times M + j.$$

A natural problem is to minimize the memory usage of the algorithm by assigning multiple instances in S to the same location. The challenge is to reclaim memory containing outdated values without overwriting *live* values. The validity of an allocation is thus closely tied with execution order: we can reuse memory cells when all the consumers of the value they contain have been scheduled.

To make this statement clearer, we need to introduce some terminology. Let $\vec{i}$ denote an instance in iteration space S. The uses of $\vec{i}$, $uses(\vec{i}) \subset S$, are the set of iteration points $\vec{j}$ such that $\vec{i} \in deps(\vec{j})$ (or equivalently, $(\vec{j}, \vec{i}) \in deps$). We wish to ensure that cell $\phi(\vec{i})$ is not overwritten before the last use of $\vec{i}$.

Let us restrict discussion to quasi-affine schedules. Such a schedule $\theta : S \to S'$ maps iteration points to their position in a new, lexicographically ordered domain S', such that $\vec{i}$ is executed before $\vec{j}$ if and only if $\theta(\vec{i}) \prec_\theta \theta(\vec{j})$. The last use of a value $\vec{i}$ is naturally the lexicographic maximum of the image of its use set by the schedule: $\mathrm{Last}_\theta(\vec{i}) = \mathrm{lexmax}\Big(\theta(uses(\vec{i}))\Big)$. Let this iteration point be $\vec{j}$. We can now state the validity condition for a memory mapping ϕ under a schedule θ:

$$\vec{i} \neq \vec{j}, \quad \vec{i} \preceq \vec{j} \preceq \mathrm{Last}_\theta(\vec{i}) \quad \Rightarrow \quad \phi(\vec{i}) \neq \phi(\vec{j}).$$

Two instances $\vec{i}$ and $\vec{j}$ which cannot be mapped to the same location are said *in conflict* (we write $\vec{i} \bowtie \vec{j}$). A schedule thus implicitly defines a set of conflicting statement instances, similar to the conflict graph used in register allocation. The analogy goes even further ; for example, whereas in register allocation, the maximum number of live values is given by the size of the maximum clique, in our context, we can define the maximum set D such as $D \times D$ is a subset of the conflicting set. The size $|D|$ gives a lower bound on required memory size.

In general, fixing a particular memory allocation early on restricts the set of valid schedules and can thus hinder, e.g., efficient parallelization. The main problem is that memory reuse introduces non-flow dependences. For this reason, it is customary to consider these two problems independently: reordering optimizations, such

[3]For simplicity, we only assume that there is a single statement S in the program ; when such is not the case, one usually makes the simplifying assumption that statements write to non-overlapping memory areas.

as tiling and parallelization, are applied first ; one then looks for an allocation that reduces memory usage, without changing the order of execution. This second step is called *memory contraction*. For simplicity, one often restricts the search to *modular* allocations, i.e., affine mappings combined with modulos with compile-time constants on each array dimension. Another approach is to use a schedule-independent memory allocations. In the specific case of stencils, one can use schedules based on *Universal Occupancy Vectors* (UOV) [52]. Such schedules are easy to compute and are not affected by execution order.

4.8 Tile Size Selection

Tile size can significantly impact the performance of a stencil implementation. Unprincipled tile size selection is likely to result in sub-optimal performance and/or waste precious resources such as bandwidth, scratchpad memory or (in the case of overlapped tiling) computing power. For this reason, various methods have been proposed to adapt the size of tiles to the problem, requirements and/or architecture at hand, usually with the goal of maximizing performance. These approaches may be classified into static (analytical) and empirical methods.

Static methods are usually based on more or less elaborate performance models. For example, early techniques may mostly consider cache/local memory size [53,54], while more recent work focuses on fine-grained modeling. Note that parametric tiling may be combined with static methods to *dynamically* select tile size based, *e.g.*, domain size parameters.

Empirical methods take the form of compile-time tuning. They are often used to adapt generic software packages to an unknown target platform. For example, the build process of several linear algebra libraries (such as LAPACK/CUBLAS) famously includes an auto-tuning phase. In this case, empirical tuning is more practical than using analytical models, as the same software may be built on a large number of (future) platforms, with possibly vastly different architectures.

4.9 Conclusion

In this chapter, we have exposed the problem of accelerating *iterative stencil computations*, a large class of compute- and data-intensive algorithms. We adopted a fairly high-level point-of-view, exposing the main challenges and techniques in a non architecture-specific way. A large part of our discussion was devoted to the tiling transformation and its variants, a tool used in most efficient implementations of stencil computations, particularly on multi-core processors Graphics Processing Units.

In the next chapter, we focus on implementation trade-offs for stencil computations on FPGAs. Their strong embedded systems roots make them a natural choice for the implementation of stencils in computer vision applications. Moreover, due to their flexibility and power efficiency compared to GPUs, FPGAs are also gaining traction in the HPC world.

Embedded and HPC applications possess dramatically different requirements: in the embedded world, one is usually interested in minimizing cost (area / power consumption) under a fixed performance constraint. In contrast, implementations targeting HPC algorithms should provide maximum performance for a given platform. Applications themselves have very different characteristics ; some stencils are much more compute-intensive than others, and grid size vary greatly between two use cases.

Unfortunately, most FPGA stencil accelerators are ad-hoc implementation, or generic templates that fail to integrate this diversity. In the next chapter, we will see that there is, in fact, no single best accelerator architecture for stencils. Based on the tiling transformation and the high-level view exposed in the current chapter, we derive a systematic methodology for FPGA stencil accelerators than can accomodate a large set of concerns.

Chapter 5

Managing Trade-Offs in FPGA Implementations of Stencil Computations

5.1 Introduction

In Chapter 4, we discussed the problem of accelerating stencil computations from a high-level, architecture-agnostic point-of-view. We showed the importance of the *tiling* transformation for both pallelizing stencils and reducing communication needs, problematics that arise on most implementation platforms.

We now focus on the design and implementation of *FPGA* stencil accelerators. Thanks to their inherent flexibility, FPGA platforms are natural choices for implementing stencils, as they can accomodate a large set of design constraints. However, we observe that earlier work on the topic has usually focused on ad-hoc implementations for specific algorithms, or failed to provide ways to enable application-specific trade-offs.

In this work, we attempt to fill this gap by providing a systematic FPGA design methodology for stencil accelerators, based on iteration-space tiling. We propose a family of designs based on sensible design parameters, exposing intuitive trade-offs between throughput, bandwidth requirements and local memory usage. We focus on system-level issues, not on fine-grained performance tuning. For this reason, we have developed a code generator producing HLS-optimized C/C++ architectural descriptions. The main design knobs are:

- Unrolling Factor: Our accelerators are based on a heavily pipelined datapath derived from HLS tools. The amount of the fine-grained parallelism at the datapath is configured through unrolling of the innermost loops. This adjusts the level of parallelism in terms of stencil update operations per clock cycle to application requirements.
- Tile Shape: The choice of tile shapes, characterized by the sizes of a tile in each dimension (possibly not tiling some of them), enables trade-offs between on-chip memory usage and bandwidth consumption.

We also propose simple analytical models for both performance and area cost to guide the exploration of the design space. Using these models, a designer is easily able to identify the most interesting design points.

We also tackle the largely ignored problem of deriving memory layouts optimized for *contiguity*. Indeed, in the kind of architectures we are targeting, memory accesses typically suffer from a significant latency overhead. Contiguous burst memory transfers must be used to achieve reasonable performance. However, typical memory layouts for stencils (such as affine modular allocations) only result in poor contiguity and do not allow for large burst transfers. We propose a data layout based on canonical projection of tile faces, partitioning inputs and outputs of a tile into a small number of contiguous regions.

This Chapter is organized as follows. In Section 5.2, we present our design space of stencil accelerators. We discuss our design principles and implementation choices, as well as our design parameters and the trade-offs they enable. In Section 5.3, we derive performance and area models for our architecture. In Section 5.4, we present our contiguity-optimizing memory layout. In Section 5.5, we present our HLS implementation and code generation flow. In Section 5.6, we present and discuss our experimental results. We discuss related work on stencil accelerators and other considerations Section 5.7, and conclude in Section 5.8.

5.2 Architectural Design Space

In this section, we present our parameterized family of FPGA stencil accelerators.

5.2.1 Target Platform

In this work we target the Xilinx Zynq platform. However, our work is equally applicable to other hybrid System on Chip platforms, such as the Intel SoC, featuring an FPGA fabric tightly-coupled with a general purpose processor. In both Zynq and SoC, the FPGA shares coherent access to main memory with the CPU cores through the last level of cache.

5.2.2 Accelerator Overview

An overview of the architecture can be seen in Figure 5.1. Our accelerator takes as input a series of tile coordinates, and computes them in a single pass. It is implemented as a bus master device on the AXI4 bus, using HP ports to access external memory. The implementation decouples memory accesses from execution through macro-pipelining at the tile-level. Macro-pipeline stages are implemented as HLS actors:

- *Communication* (*Read* and *Write*) actors read/write tile results from/to main memory through HP ports.
- The *compute* actor performs actual tile computations.

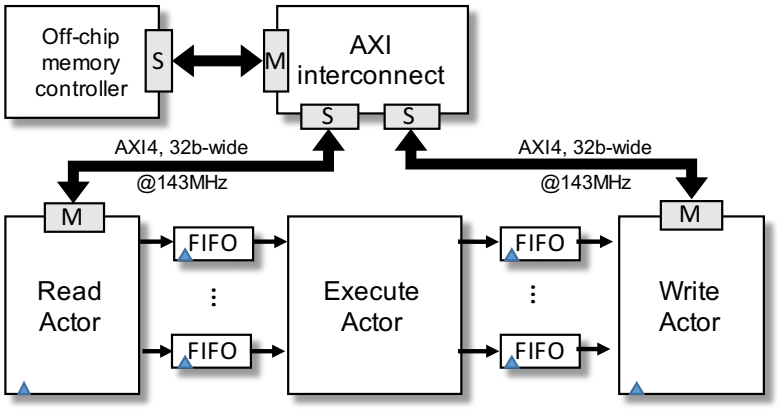


Figure 5.1: Diagram of the architecture.

Communication and compute actors are inter-connected with FIFOs of sufficiently large size. The main benefit of this decoupling is that it provides overlapping between computation and communication: tile inputs can be fetched / committed in parallel with computation. If available bandwidth is sufficient, memory access time can thus be effectively hidden, provided that the number of tiles to execute is large. Indeed, even with computation/communication overlapping, the execution of the first and last tile in a sequence necessarily introduces some latency overhead.

5.2.3 Compute Actor

We now focus on the compute actor, the computationl core of our architecture.

Execution Datapath

The compute actor computes each tile in a single sweep using a deeply pipelined datapath, derived by te HLS tool. The entire set of operations in a tile is pipelined with Initiation Interval of one. In terms of input C code to HLS, this pipelining is realized by coalescing the loop nest iterating over a tile into a single loop, and pipelining the body of the loop.

The updates within a single time step are independent of each other and can be fed to the datapath every cycle, provided that the data is available on FIFOs. However, pipelining *across* time steps requires that results from the previous time step have been entirely computed before being first accessed, which is not necessarily true in presence of pipelining. This imposes a constraint on tile size and pipeline depth ; such constraints are discussed in Section 5.2.5.

Our datapath can be further configured to perform an arbitrary number of stencil updates per cycle, simply by unrolling the innermost loop by a fixed factor before coalescing. Adjusting this factor allows us to control the computational intensity of our IP. Empirically, we observe that pipeline depth Δ depends on the target operating frequency provided by the user, but *not* on unrolling factor. It is not surprising, since propagation time should not be affected by the replication induced by unrolling, but is a beneficial property for controlling throughput.

Memory Re-Use

The execute actor takes advantage of reuse of input data and intermediate results within a tile. We apply a technique similar to the one by Cong et al. [55] to minimize local memory usage and avoid memory bank conflicts. However, we use HLS arrays instead of explicit FIFOs, which necessitates dealing with the initialization of these arrays for each tile. This can be achieved in two ways:

1. By increasing the number of memory ports in the on-chip memory to parallelize the initialization (i.e., without performance overhead)
2. By inserting wait-states to serialize the initialization phase.

We use the latter as on-chip memory is a scarce resource. These wait states correspond to the halo of the tile: our loop scans halo regions along with iteration points to pre-load external dependencies. If such were not the case, for example, the first iteration on a tile would need to read all its dependencies in parallel from different BRAMs or FIFOs. With this approach, all external dependencies are already available in registers or on-chip memory when an update is started.

5.2.4 Overview of the Read/Write Actors

The read actor streams in input data to the execute actor, and the write actor streams out results from the execution actor. These actors perform burst accesses to external memory through the AXI4 interface. We use a custom data layout, discussed in Section 5.4, to ensure that most memory accesses are contiguous.

We take special care to minimize idle time and maximize bus occupation to get as close as possible to the maximum achievable bandwidth (*e.g.*, 600 MB/s per HP port with 32 bit bus width). To this aim, the compute actors are in fact split in several parallel actors to, *e.g.*, re-order memory elements and compute addresses in parallel with actual memory transfers.

5.2.5 Design Parameters

Our approach aims at exposing relevant design knobs to drive the design space exploration. These knobs are:

- The choice of the Unrolling Factor (UF), representing the number of stencil updates per cycle (in steady state). The datapath performs UF updates parallel. The value of UF hence determines the amount of parallelism. Increasing this factor will boost the maximum throughput that can be attained, but will also raise bandwidth requirements to keep feeding the datapath.

  The choice of this parameter is mostly driven by the throughput requirements. Larger values give higher throughput, but increase area cost. In HPC applications, one will want to increase this factor as much as possible, while in many embedded applications, a small value of UF may be sufficient.

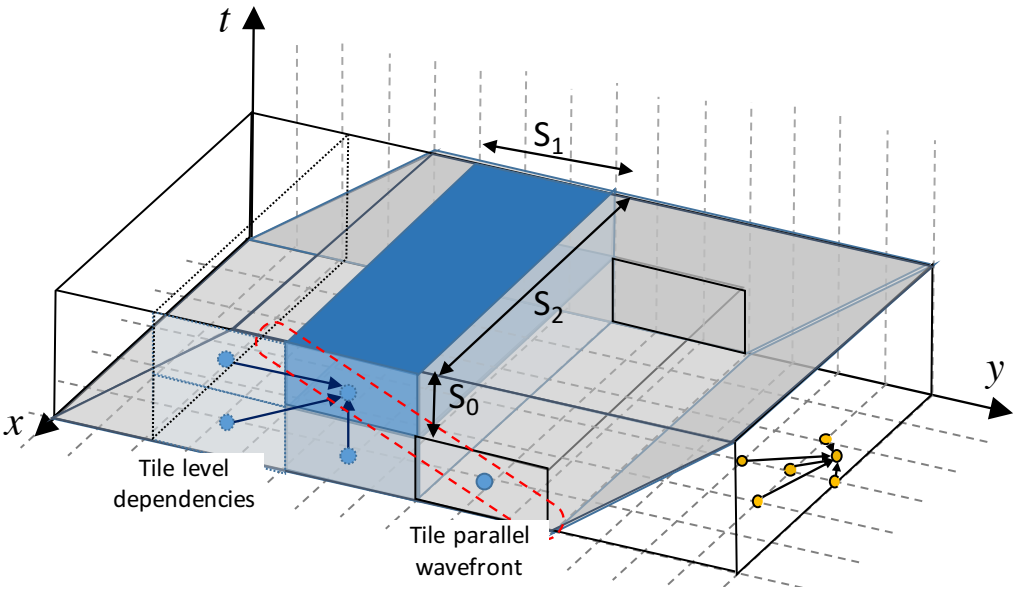


Figure 5.2: Illustration of partial tiling for a 2D jacobi.

- The choice of the tile sizes $(S_0, \ldots, S_d)$ is also critical, as tile shape determines data locality. Tile sizes control the trade-off between off-chip bandwidth requirements and on-chip memory usage. Larger sizes reduce bandwidth requirements, but increase on-chip data storage.

  Larger tile sizes also reduce the overhead due to halo regions, further improving throughput.

- Finally, we allow for *partial tiling* (see Figure 5.2): one may choose to only tile outer domain dimensions, while leaving one or several inner dimensions untiled. In these dimensions, a tile spans the entirety of the domain. This technique can be interesting if the stencil grid is too large to fit in on-chip memory, but bands across only some dimensions are not. Another benefit of this approach, compared to full-tiling, is that it reduces the overhead of partial tiles, due to only a subset of domain dimensions needing to be skewed.

  While tiling in all dimensions can be used for domains of any (and even unknown) size, partial tiling is only applicable if domain dimensions are known at compile-time and relatively small. For this reason, it is an interesting approach, for example, for computer vision algorithms, while tiling all dimensions may always be required in most HPC use-cases.

Constraints We require that tile size in the innermost dimension, S_d, is evenly divisible by UF. This constraint avoids complex controls arising from cases where only a subset of unrolled iterations are valid computations. Also, Tile sizes in the spatial dimensions are constrained to have more iterations than the pipeline depth: $S_1 \times \frac{S_2}{\text{UF}} > \Delta$ to prevent dependence violations.

5.3 Performance Modeling

The parameters above expose a huge design space to be explored. In this section we present performance models to guide the exploration of this space. We adopt the following conventions:

- The dimensions of a tile in the skewed iteration space are written as $S_0 \times S_1 \times \ldots S_d$, where d is the number of *data* dimensions. S_0 is thus the number of time steps spanned by a tile, while for $i > 0$, S_i represents its extent along the i-th data dimension. In case of partial tiling, dimensions $S_{d-k+}, \ldots, S_d$, where k denotes the number of untiled dimensions, correspond to iteration domain dimensions.
- Dependence vectors are denoted $(d_0, \ldots, d_{n_d-1})$, where n_d is the number of such dependencies. For any $i \leq d$, u_i is the i-th canonical basis vector of $\mathbf{Z}^{d+1}$ with 1 as its i-th component and 0 elsewhere.
- The unrolling factor is written UF.

5.3.1 Asymptotic Performance Model

The important metric to model is the number of stencil updates per cycle[1], computed as follows:

$$\text{UpdatesPerCycle} = \frac{\text{TileVolume}(S_0 \times S_1 \times S_2)}{\text{TileCycles}}.$$

TileCycles denotes the number of cycles it takes to execute a tile, while TileVolume $= \Pi S_i$ simply corresponds to the amount of computation in a tile.

Assuming overlapping between computation and communication, TileCycles corresponds (in steady-state) to the slowest between the communication and computation tasks. In other words, is the maximum between the number of cycles spent for computing, CompCycles, and the number of cycles spent for memory transfers, CommCycles:

$$\text{TileCycles} = \max(\text{CompCycles}, \text{CommCycles})$$

This is the asymptotic performance of our design that is reached when the problem size is large enough to make the overhead at the boundaries (where the computation and communication are not fully overlapped) negligible.

Performance of the Compute Actor

Recall that the pipelined datapath of the compute actor computes, in steady-state, UF updates per cycle. In addition to the tile volume, the compute actor scans the boundary *halo* regions to fetch input data. Representing the dependence depth in the d-th dimension (*i.e.*, the thickness of the halo in that dimension) as h_d, the number of times the compute actor datapath is invoked per tile is thus:

$$\text{CAVolume} = S_0 \times (S_1 + h_1) \times \left\lceil \frac{S_2 + h_2}{\text{UF}} \right\rceil$$

[1]Note that UpdatesPerCycle is a direct proxy to throughput, which is UpdatesPerCycle $\times$ FlopsPerUpdate $\times$ Frequency.

Since the initiation interval is always 1 for our design, the total number of cycles that it takes to execute the compute actor, assuming all inputs are ready, is given by:

$$\text{CompCycles} = \text{CAVolume} + \Delta - 1$$

where Δ denotes the pipeline depth of the compute actor datapath. Pipeline depth, determined by the HLS tool during RTL generation, is a function of the update formula and synthesis frequency, but is not influenced by the choice of tile sizes or unrolling factors.

Communication Modeling

It is critical to make use of burst communication to maximize bandwidth utilization. Indeed, the number of cycles for a memory transfer of n contiguous words can be accurately estimated as:

$$\text{BurstCycles}(n) = n + \text{BurstLatency}$$

$$\text{BurstLatency} = \text{Frequency(MHz)} \times K_{\text{burst}} \times 0.01$$

where K_{burst} is a constant representing burst latency at $100MHz$ (about 30 cycles in our case). For this reason, we use a custom memory layout, detailed in 5.4, to ensure that almost all memory transfers permit burst accesses. Moreover, we concurrently use all HP ports such that burst latency can be totally hidden. Hence, modeling the communication cost can be simplified to modeling the data volume.

The data volume to be communicated is exactly the halo regions of a tile. This can be approximated by:

$$\text{CommVolume} = \prod_{i=0}^{d} (S_i + h_i) - \prod_{i=0}^{d} S_i.$$

When the data element is one word, CommVolume directly translates to the number of transfer cycles: CommCycles. If the stencil operates, for example, on multiple numerical fields, this formula may need to be changed to reflect larger element-size.

5.3.2 Modeling the Area Cost

Precise modeling of the area cost can be extremely challenging, and is heavily influenced by the HLS tool and the algorithm. However, it is not difficult to make relative comparisons among design points in our parameter space.

Indeed, we can expect unrolling factor and communication volume to both have linear relationships with area: UF with LUTs/DSPs and communication volume with on-chip buffer requirement. This suggests, prior to Design Space Exploration, to sample the design space and perform linear regressions in order to compute area models.

To aggregate usage reports into a single number, we choose to use the sum of the utilization rates (Slice/BRAM/DSP) as area metric. While no metric is perfect, we

choose this one as it captures the relative scarceness of hardware resources on the board. To represent the interaction between UF and tile size, and to relate these values to area, we propose infer the following linear functions from a few design points:

$$C_{\text{dp}} = a_{\text{dp}} \times \text{UnrollFactor} + b_{\text{dp}}$$
$$C_{\text{mem}} = a_{\text{mem}} \times \text{CommVolume} + b_{\text{mem}}$$

where $a_i,\ b_i$ must be learned with linear regression.

As we demonstrate in Section 5.6, a simple estimate based on unrolling factor and tile face volumes gives sufficient insight about the area cost to guide the design space exploration. Moreover, we will see that only a few design points are required to obtain sufficiently accurate models, which means that this approach is practical.

5.4 Data Layout

We mentioned in Section 5.3.1 that contiguity plays an important role in maximizing memory bus occupation, as it reduces access latency thanks to burst accesses. Indeed, only a small fraction of the available bandwidth can be effectively used without bursts. For example, on the ZC706 board we use for our experiments, each memory access incurs a latency penalty of about about 30 cycles. This means that the best bandwidth efficiency that can be achieved on a given memory port, using bursts of maximum size (256 words) is approximately 90%. However, if we instead use bursts of only 32 words, we see efficiency drops to about 50% because of burst latency. It is thus critical for performance that most communication is performed using bursts of maximum size.

Traditional memory layouts for multi-dimensional arrays (and modular contractions thereof, such as alternating between two copies of spatial grid) typically provide contiguity outward from innermost array dimensions: for example, in a 3D array `A[M][N][P]`, consecutive elements `A[t][x][y]` and `A[t][x][y+1]` may be stored contiguously, as well as **full** consecutive lines `A[t][x][_]` and `A[t][x+1][_]`.

Consequently, when tiling iteration spaces along all dimensions, with such layouts, we find ourselves repeatedly accessing many small contiguous fragments. For example, the rectangular region `A[t][x][y]` $\rightarrow$ `A[t][x+N][y+M]` is composed of $N + 1$ contiguous segments of length $M + 1$. Suppose that M and N represent tile dimensions, and that this region represents the halo-face of a tile along the temporal dimension. M and N are likely much smaller than maximum burst size, and communication time is thus dominated by burst latency.

The discussion above suggests to partition the inputs set of a tile into faces, and to store those faces contiguously. It is easy, using this kind of decomposition, to ensure that each face can be *read* contiguously. In fact, it is equivalently easy to enforce that all tile inputs form one contiguous segment in memory, for example by ordering them by their first read in the tile. The problem is that, then, *writes* cannot be performed contiguously, and nothing is gained performance-wise because of this asymmetry. Moreover, some results may be used by multiple tiles and must hence

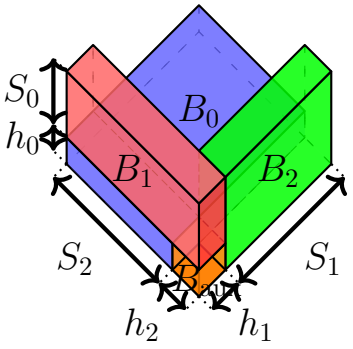


Figure 5.3: Input faces of a tile.

be stored redundantly ; there is thus an imbalance between read and write volume. Of course, we can also ensure contiguity for writes, with symmetric problems.

We tackle this challenge by storing projection of tile inputs / outputs along each dimension into distinct buffers. In this section, we present an allocation strategy providing the following benefits:

- Each tile can read/write all its inputs/outputs in a small number of contiguous accesses, usually providing enough contiguity that burst latency becomes negligible.
- Read and write volume are strictly equal, with some outputs being stored in multiple buffers.

Our idea is to project tile faces along canonical dimensions, splitting communication volume into distinct buffers such that both tile inputs *and* and outputs map to contiguous segments in these buffers.

5.4.1 Example: 3D Iteration Space

Our decomposition is illustrated in Figure 5.3 for a 3D iteration space. Inputs to a tile are partitioned into 4 buffers, B_0, B_1, B_2 and B_{aux}. The "thickness" of each buffer $B_0, \ldots, B_2$ corresponds to dependence depth h_d along that dimension. Each buffer is extended along exactly one dimension ; this is necessary to enforce contiguity, and necessitates the introduction of buffer B_{aux}, of size $B_0 \times B_1 \times B_2$ to store the "corner" of the tile.

Observe that each input corresponds to a full face from a neighboring tile, and part of a face from a diagonal neighbor. For example, let (t, x, y) denote current tile coordinates ; buffer B_2 corresponds to outputs of tiles $(t, x, y-1)$ and $(t-1, x, y-1)$. It means that we must be able to "shift" a window on the axis along which this face is extended, and that the corresponding regions must always correspond to contiguous memory segments. This requirement imposes that all input/output buffers B_i are stored in a single array. Also, it constrains the order of dimensions within these arrays: the projection-dimension i must be the innermost, and the one along which we are extending the outermost.

In this example, we have chosen to extend dimension tile face 0 along dimension 1, tile face 1 along dimension 2 and tile face 2 along dimension 0. Dimensions order

for arrays B_0, B_1 and B_2 are thus:

$$B_0 : 1, 2, 0 \quad B_1 : 2, 0, 1 \quad B_2 : 0, 1, 2$$

Since $h_0 = 1$ in many stencils, effective order of dimensions in buffer B_0 is usually canonical ; however, in buffer B_1, array dimensions are "rotated" compared to natural order $0, 1, 2$. It means that, while inputs can be read/written contiguously, they must unfortunately be buffered on-chip as the order in which they are used (produced) by a tile does not match the order in which they are read (written) to/from main memory.

Finally, as with most memory allocations (see Section 4.7), one of our goals is to reduce memory consumption by re-using memory cells. This can be easily done at the face-buffer level using modular allocation.

5.4.2 Generalization

A natural question is to determine whether the allocation strategy presented above for 3D iteration spaces can be extended to higher-dimensional cases. A first attempt is to replicate the partitioning above by similarly extending each face along exactly one dimension, and using one auxiliary buffer B_{aux} of size $\prod h_i$. However, there are some issues with this approach.

Indeed, let $X = (x_0, x_1, \ldots, x_d)$ be the vector of coordinates of a tile. Suppose that face B_i is extended along dimension j. Then, B_i contains points from tiles $X - u_i$ and and $X - u_i - u_j$, where u_k are unit vectors. B_{aux} contains dependencies from tile $X - \sum u_k$. However, by simply extending face along one dimension, we miss dependencies from diagonal tiles distant from current tile by more than 2 and less than $d+1$ unit vectors. For example, in a 4D iteration space, we might typically miss dependencies from tile $(x_0 - 1, x_1 - 1, x_2 - 1, x_2)$

Generalizing our memory layout to higher-dimensional spaces hence requires the introduction of additional buffers to handle such "corner" and "edge" cases and cover all dependencies. For a $4D$ iteration space, 3 additional buffers of size $S0 \times h_0 \times h_1 \times h_2, \ldots$ must be introduced. Overall, we thus need 8 distinct memory arrays.

At the time of this writing, we have only implemented our memory layout for the 3D iteration space case, and further work is required to properly handle higher-dimensional cases.

5.5 Implementation

Our family of accelerators has been implemented via a code generator producing C/C++ architectural descriptions. This code is optimized for HDL synthesis by Vivado HLS, and system integration (block diagram generation, hardware invocation) is handled automatically by SDSoC. The output of our generator thus consists almost exclusively in C/C++ code, for both the hardware and software parts.

We did not choose to produce HDL (VDHDL or Verilog) descriptions because (i) fine-tuning for performance was not our primary goal (ii) HLS tools are now mature enough that system-level issues impact performance more significantly than

missed micro-optimizations. In fact, we believe that targeting a high-level language and relying on (pragma-driven) HLS optimizations significantly increased our productivity and the quality of our design. For example, the ability of Vivado HLS to derive multiple-input, multiple-output deeply pipelined operators (*e.g.*, 87 stages for anisotropic diffusion) is essential to our methodology.

5.5.1 Code Generator Overview

Our experimental toolchain has been made freely available as open-source software[2]. Current implementation is specialized to 2D stencils operating over 3D iteration spaces, and implementations of full and partial tiling are distinct. However, there is no fundamental reason for doing so and much of the code is in fact shared between both implementations ; some engineering work is required to generalize these implementations into a single generic tool.

The input of the generator are the tile size, the unroll factor and the update formula. In the case of partial tiling, tile size in the last dimension corresponds to domain size. All other domain dimensions (for both full and partial tiling) are left as runtime parameters.

Based on these inputs, the generator produces:

- A software program, in charge of allocating/initializing memory buffers and orchestrating the execution of tile wavefronts.
- HLS code for the IP.
- A `Makefile` and synthesis scripts.

The `Makefile` can be used to compile the program to software code and for simulation and validating the implementation against a reference (untiled) software implementation. It can also be used to invoke the synthesis scripts and generate a bootable SD card for the Zynq board. This SD card contains a bitstream with our IP, a minimal Linux distribution and a version of the generated program invocating our IP for the computation of tiles.

Our code generator is implemented as a set of Python scripts. Generation of the code for scanning tile wavefronts is handled by the *Integer Set Library* [56]. The HLS code is mostly based on a template.

5.5.2 HLS Code Overview

It is well known that, while the efficiency of HLS tools has significantly improved over the years, HLS-optimized code written by a seasoned hardware designer is still very different from functionally-equivalent code written by a software programmer. The goal of this paragraph is to give the reader a hint of some of the challenges and code changes we had to implement to reach reasonable performance. Our guiding principle was to bring the performance our IP as close as possible to the theoretical ideal of the roofline model. This was done by exploiting parallelim at all-levels to hide latency and by avoiding at all costs the introduction of idle states in the Read/Write/Compute processes.

[2] `https://github.com/gdeest/hls-stencil`

```
void top(int N, T* arr_in, T* arr_out) {
  #pragma HLS DATAFLOW

  fifo_t fin, fout;

  // Read Actor
  for (int i=0; i<N; i++) {
    #pragma HLS PIPELINE II=1
    fin.write(arr[i]);
  }

  // Compute Actor
  for (int i=0; i<N; i++) {
    #pragma HLS PIPELINE II=1
    fout.write(compute(fin.read()));
  }

  // Write Actor
  for (int i=0; i<N; i++) {
    #pragma HLS PIPELINE II=1
    arr[i] = fout.read();
  }
}
```

Figure 5.4: Use of the `DATAFLOW` directive to implement computation / communication overlapping.

Our architecture is realized as a set of HLS actors, implemented with the `DATAFLOW` directive. An extremely simplified architecture skeleton representing the use of this directive is illustrated in Figure 5.4. With this pragma, the HLS tools automatically infers task-level parallelism within a basic block, and implements tasks as independent HDL processes communicating through FIFOS or ping-pong buffers. Unfortunately, this directive presents several limitations:

- First, it is inherently oriented toward streaming access patterns and does not allow for feedback loops, which unfortunately prevents us to use it for handling reuse within a tile[3]. While our code could have benefitted from actor-based parallelism at different levels, we must restrict its use to the macro-pipeline of tiles.
- Secondly, it is not reliable for inferring useful parallelism when communicating through arrays, unless access patterns are extremely regular and the tool can prove that each element is read/written exactly once. Since we must deal with complex guards to handle tile boundaries, our code breaks this analysis and we must use explicit FIFOs for communication between actors.

Communication and computation are thus performed by independent processes to implement computation / communication overlapping. We rely crucially on the ability of the HLS tool to infer burst accesses when using pipelined loops to read-/write from/to AXI4 bus master interfaces. Using this implicit mechanism, instead of explicit bursts performed with the `memcpy()` function (natively recognized by Vivado HLS), allowed us to reliably stream values to FIFOs in parallel with memory accesses.

Full tiling uses a custom, tile-face based memory layout (see Section 5.4). Accesses between different buffers are in fact performed in parallel by different read-/write actors. Since some faces cannot be read/written in lexicographic order via burst accesses, we use additional HLS actors to re-order these values. This introduces additional re-ordering buffers, and thus constitutes a "hidden" cost of full tiling. Note, however, that this overhead is indirectly taken into account when deriving area models through linear regressions.

Since the `DATAFLOW` directive does not allow for loops in the actor graph, the execution datapath of the compute actor cannot be kept as "clean" as we would like. It actually performs three different tasks:

- Computing results.
- Reading inputs from input FIFOs (through additional iterations) and writing outputs to output FIFOS (when relevant).
- Managing reuse memory.

Our memory architecture for storing inputs and intermediary results is similar to that proposed by Cong et al. [55], with one memory buffer per iteration space dimension.

[3]This limitation is necessary to allow semantically equivalent sequential software simulation of the IP. In our opinion, it illustrates one limitation of pragma-based HLS: with a high-level description of the architecture using native software constructs, this limitation could be overcome.

Table 5.1: Number of floating-point operations, and pipeline depth for one update of the kernels.

Kernel	flops	Pipeline depth
Jacobi 2D	1 ×, 4 +	43
Anisotropic Diffusion	9 ×, 17 +, 2 /, 9 exp	87

Finally, to minimize the number of memory ports used and overall simplify control, we implement a unrolled stencil as a stencil operating over a wider data-type. This allows to read and write from at most one-FIFO per loop iteration, and significantly simplifies some analyses by the HLS tools.

5.6 Experimental Validation

We proceed with the experimental validation of our methodology. We emphasize that our motivation is *not* to derive the design with the highest throughput ; indeed, we wish to accomodate different situations with specific performance requirements. We need to show that we are able to select the "right size" for a given context. In this section, our goal is thus:

- To establish the accuracy of the performance model for different design parameters.
- To show that design points can be successfully compared in terms of hardware resource usage.

5.6.1 Experimental Setup

Kernels

We validate our work on two different stencil kernels: Jacobi 2D and Anisotropic diffusion. Jacobi 2D is a standard example for stencils that have relatively few number of operations, such as the heat equation, and is strongly bandwidth constrained. Anisotropic diffusion is an iterative smoothing filter, which is much more compute-intensive. The characteristics of their update operations are summarized in Table 5.1.

Tools / Platform

We use our code generation (see Section 5.5) and Xilinx SDSoC 2016.3. Designs are synthesized for the ZC706 Zynq evaluation board (featuring an XC7Z045 chip). The target frequency for all designs was set to 142.86 MHz (*i.e.*, the maximum frequency supported by default by SDSoC which is below the architectural limit of 150 MHz for the AXI4 bus). We could have used multiple clock domains to synthesize the compute actor at a higher frequency that the communication actors ; however, this is not currently natively supported by SDSoC, and would have required dropping to Vivado HLS and performing the integration manually in IP integrator. Since

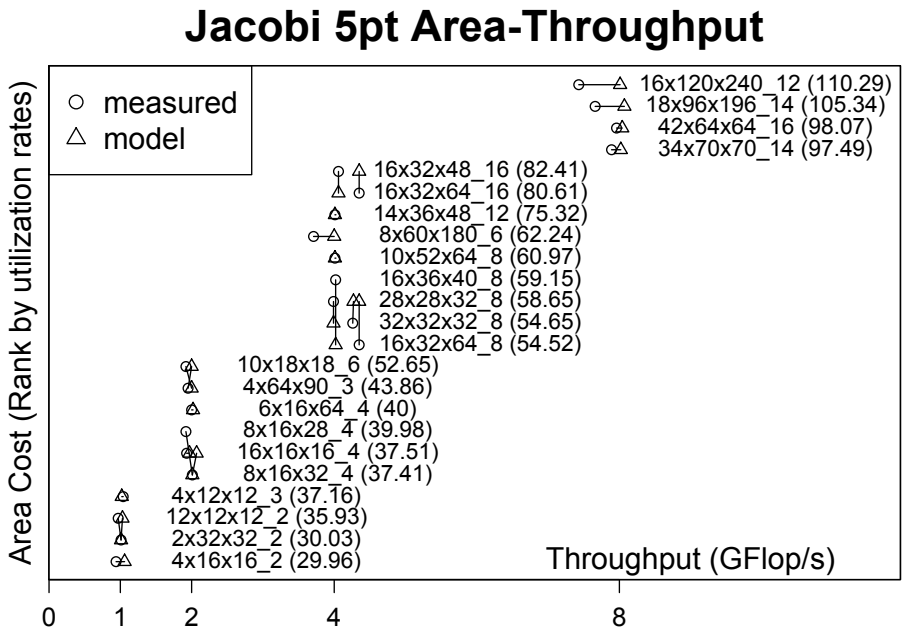


Figure 5.5: Predicted and measured area/throughput for the Jacobi2D kernel.

(i) our goal is to validate our methodology as a whole, rather than fine-tuning the hardware for better performance, and (ii) this would likely provide only minor frequency improvements, we believe this compromise to be benign. We note that extending our models to the case where compute and communication actors are synthesized at different frequencies is straightforward.

Methodology

For each kernel, we used our code generator (see Section 5.5) to generate a series of design using different tile sizes, unrolling factors and tiling modes (full and partial). These designs were then synthesized using Xilinx SDSoC 2016.3, targeting the ZC706 Zynq evaluation board with an XC7Z045 chip. For each design point, we sampled the number of CPU cycles it took to execute a set of tiles on the board with the Zynq Global Timer. This timer represents a number of CPU cycles, which was than converted to FPGA cycles based on their erlative frequency (the default clock frequency for the ARM cores is 800 MHz).

Contrary to many prior work, all performance numbers provided in this section were obtained from actual accelerators instances running on the target FPGA platform. Hence, our results account for all performance degradation issues related to bus interconnect and/or external memory.

We adopt the following convention: a design is abbreviated as S_0xS_1xS_2_UF. As mentioned in Section 5.3.2, the area cost is the sum of utilization rates for Slices/BRAMs/DSPs, and takes a value between 0 and 300. The target board has 54650 slices, 545 BRAM tiles, and 900 DSPs.

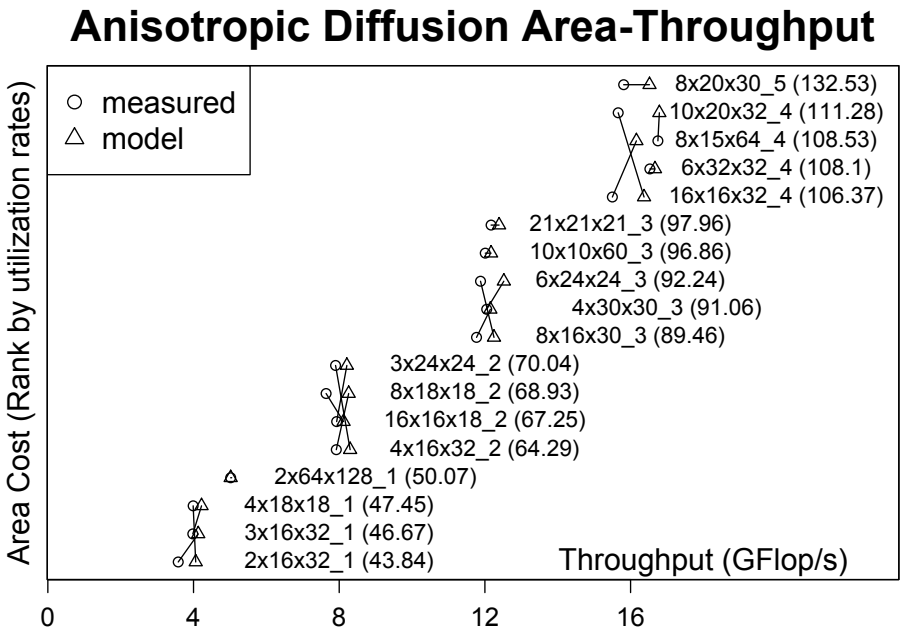


Figure 5.6: Predicted and measured area/throughput for the Anisotropic Diffusion kernel.

5.6.2 Full Tiling

Jacobi 2D

We use four target performances; 1, 2, 4, and 8GFlop/s; to illustrate the trade-offs exposed by our design knobs. A number of design points that have the desired performance with different tile shapes were synthesized. Linear regression for the area model used the following four design points: 4x16x16_2, 8x16x32_4, 32x32x32_8, and 64x64x64_16.

Figure 5.5 summarizes the area and throughput of the resulting designs, as well as those predicted by the model. Throughput is represented in the horizontal axis, while are is shown in the vertical axis. For readability reasons, area is displayed as *rank*: design points are vertically ordered from the less costly to the most expensive in terms of resource usage.

One thing that is clearly visible in the figure is that the performance model is quite accurate. Almost all points are on the target GFlop/s based on the model. Moreover, performance results were extremely stable across repeated experiments. This is not necessarily surprising, as FPGAs are mostly deterministic ; however, bus contention issues and access to shared memory could have introduced significant variability, which was not the case. The largest divergence from model is 7% (16x120x240_12) ; as these divergences were also highly reproducible, this suggests that the performance model could be further improved.

The area result is also mostly in agreement with the model. An unexpected event was the observation of some interchanges between the ranks predicted by the linear area model. It was found that these differences were due to powers-of-two tile sizes leading to much simpler address computations (especially modulo operations)

Table 5.2: Resource Usage for the Jacobi 2D kernel

Design	Slices	BRAMs	DSP48E
$2 \times 32 \times 32$, UF=2	9663	27	66
$8 \times 16 \times 32$, UF=4	11123	30	104
$16 \times 32 \times 64$, UF=8	13148	57	180
$34 \times 70 \times 70$, UF=14	18103	126	372

in communication actors, and thus using less DSPs. This favors powers-of-two tile sizes over slightly smaller shapes using less on-chip memory. However, it is mostly an artefact of the HLS tool: since tile sizes are known at HDL-synthesis time, these operations could be implemented more efficiently in FPGA logic, probably reducing or eliminating the occurrence of such phenomena.

To minimize area cost, the design knobs we provide suggest to first set the unrolling factor to the smallest value that can meet performance requirement, and then adjusting tile size to "feed" the datapath at a sufficient rate. However, we observe that in some use cases, using slightly higher UF may be beneficial for area cost. Design points such as 8x60x180_6 and 16x120x240_12 are examples of these cases. We might explain this phenomenon by the diminishing returns from increasing the tile sizes. Indeed, increasing tile size improves performance in two different ways:

- by improving locality ;
- by reducing the overhead due to the halo regions.

Once the tile size is large enough to keep the datapath busy (i.e., the accelerator is no longer I/O-bound), then further performance comes only from reductions in overhead. The above designs are in such situations, where the performance target is at the limit of what can be achieved by the given UF, such that large tile sizes have to be used to meet the goal. Our performance model can identify these situations and point to better design points.

We report the resource usage for the best performing designs for each target performance in Table 5.2.

Anisotropic Diffusion

We use four target performance levels: 4, 8, 12, and 16 GFlop/s. A number of design points that have the desired performance with different tile shapes were synthesized. The area model is learnt with the following points: 2x16x32_1, 4x16x32_2, and 16x16x32_4. The area-throughput trade-off is summarized in Figure 5.6, and Table 5.3 reports the detailed resource usage for the best performing designs.

We do not repeat the same discussion as in Jacobi 2D case; all of them applies to anisotropic diffusion as well. One key difference is that the importance of BRAM is much less significant compared to Jacobi 2D. This is because the arithmetic intensity of this kernel is high (37 floating-point operations, including 9 exponentiation), and not much data locality is needed to keep the accelerator busy.

Table 5.3: Resource Usage for the Anisotropic Diffusion kernel

Design	Slices	BRAMs	DSP48E
$2 \times 16 \times 32$, UF=1	12675	29	138
$4 \times 16 \times 32$, UF=2	17119	30	248
$8 \times 16 \times 30$, UF=3	22961	32	375
$16 \times 16 \times 32$, UF=4	26000	37	468

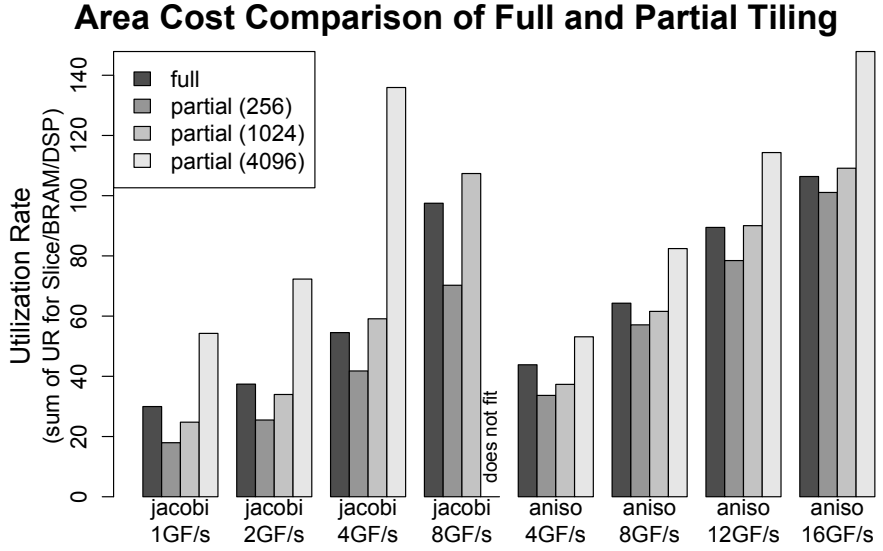


Figure 5.7: Area results of partial tiling for the two kernels and different performance targets.

5.6.3 Partial Tiling

Partial tiling is an attractive alternative to full tiling for small problem sizes. To illustrate this, we implemented this strategy and compared it to fully tiled cases. Figure 5.7 illustrates the trade-offs between the two approaches.

We selected the less expensive design meeting different performance targets for both full and partial tiling. In the case of partial tiling, observe that area cost actually depends on domain size, since tiles span an entire dimension of the domain. Consequently, only one result is reported for full tiling, while in the case of partial tiling, several designs were generated for different domain sizes.

For both kernels, observe that partial tiling always provides lesser area cost for small problem size, but the situation is reversed for larger domains. This can be explained by the growing buffer requirements for partial tiling This reversal also occurs sooner as performance requirement increase, since larger tile size must be used ; partial tiling offers less degree of freedom in adjusting tile size to increase compute/IO ratio. Partial tiling is thus beneficial for relatively small problem sizes with moderate performance requirements.

Note that, for anisotropic diffusion, it scales to larger problem sizes for a given performance target. This is a consequence of choosing GFlops as a performance

metric: since anisotropic diffusion is much more compute-intensive than Jacobi, more operations are performed per update, and thus, per byte transferred from main memory. Smaller tile sizes can be kept to meet a given GFlops target.

5.7 Discussion

In this section, we conclude with a discussion on earlier work and some additional considerations.

5.7.1 Comparison with Earlier Work

The main contribution of our work is a family of designs that cover wide range of performance requirements, accompanied by a performance model to quickly narrow down on the most relevant design points. It would be interesting to directly compare our design with earlier work experimentally. However, this is not practical. Indeed, most implementations are not freely available, and often target different FPGA platforms. Fortunately, a large subset of prior work is within, or comparable to, the family of designs described in this chapter. We may hence compare their relative system-level characteristics.

Untiled variants [55, 57, 58] may be viewed as designs with tile size in the time dimension set to 1, and the remaining dimensions equal to the problem size. They can give similar performance to other designs for very small problem sizes, where the entire data fit on chip ; however, for larger instances, intermediary results must be "spilled" to main memory and temporal locality cannot be exploited. To demonstrate the ineffectiveness of such approaches compiled to tiled variants, we have implemented untiled stencils. Attaining 1GF/s with Jacobi 2D kernel for 256×256 image uses 25% of the available BRAM, and the limit is reached with 512×512 using 95% of BRAMs.

Natale et al. [59] propose to scale computations of iterative stencils to multi-FPGA systems by replicating and chaining *Streaming Stencil Timesteps* (SST) over several FPGAs. Each SST is in charge of computing one timestep over the entire grid. Within a timestep, intermediary results are kept on FIFOs of optimal size, but are also forwarded to the next SST. Compared to our design, this approach provides only limited control over throughput and bandwidth requirements. Throughput may be increased by increasing the number of SSTs ; however, this leads to a corresponding increase in local memory requirements. One may view this strategy as a combination of (i) temporal tiling with (ii) an unrolling of *the outermost loop dimension*. Chaining multiple FPGAs allows this architecture to scale to arbitrary throughput requirements, but the absence of tiling in spatial dimensions means that it cannot handle arbitrary problem sizes.

We did not implement overlapped tiling, because the overhead due to redundant computations is in most cases too significant, as revealed by a quick analysis. For example, consider the Jacobi 2D example, where the data dimensions are tiled by $S \times S$. Overlapped tiling for two time steps requires the halo of $S \times S$ extend by one in each direction to be redundantly computed. The amount of redundant

computation grows as tile size along the time dimension is increased. Generalizing to any tile size $(S_t + 1)$ in the time dimension, the overall computational volume is :

$$\sum_{x=1}^{S_t} \left((S + 2x)^2 - S^2 \right) = \sum_{x=1}^{S_t} \left(4Sx + 4x^2 \right)$$

which can be further simplified to:

$$4 \left(\frac{S_t(S_t + 1)}{2} S + \frac{S_t(S_t + 1)(2S_t + 1)}{6} \right)$$

The relative importance of the overhead depends on the non-redundant computation $(S_t \times S^2)$. When $S_t = S$, the overhead is more than three times larger than the actual computation (367 %). Tiles must hence be kept thin and flat to reduce this overhead ; but doing so also decreases temporal re-use, and is not interesting on our platform.

5.7.2 Additional Considerations

We have extensively discussed the trade-offs of different design choices in Section 5.6. Many other factors, if taken into account, could also influence the trade-off, such as synthesis frequency of the different actors. In this work, all designs were synthesized at the same frequency (143 MHz) on a single clock domain. The communication and computation parts could use independent clocks, allowing the compute actor to reach higher frequencies than the 150 MHz limit of the AXI bus. This change would unveil another design dimension that should be considered when selecting a design for a particular context.

We have not discussed how to handle domain boundaries. In our implementation, boundary and incomplete tiles are currently handled in software, in parallel with "full" tiles in the same wavefront. For small domains and/or large tile sizes, this can introduce a large performance overhead as the software is much slower than the accelerator, and the number of "full" tiles is low: the software then becomes a performance bottleneck. Moreover, steady-state performance is not attained on small tile wavefronts.

This problem is not fundamental, as the domain could be easily padded with "dummy" iterations to execute all tiles in hardware. Padding has an impact on overall performance that depends on tile size, problem size, and tiling strategy. For example, we would have 10.09% dummy iterations with $4 \times 16 \times 16\_2$ tiles on a $50 \times 512 \times 512$ domain. However that this overhead decreases with partial tiling for the same performance target, and drops to 2.63% with $2 \times 16 \times 512\_2$ with the same domain size.

In this work, we only considered single-field stencils with Jacobi-style dependences operating on 32-bit floating-point data. Our generator could be easily extended to Gauss-Seidel dependence patterns (with an additional skewing) and multi-field stencils such as FDTD. The use of fixed-point or custom floating-point arithmetic would open a whole avenue of trade-offs involving accuracy, giving FPGAs a real advantage compared to less flexible platforms such as GPUs.

All these factors influence throughput and/or area, and could impact the specific trade-offs and performance modeling, by introducing additional design knobs.

5.8 Conclusions

In this chapter, we discussed a design methodology for FPGA stencil accelerators based on a tunable family of designs and accurate performance / area models. Our results show that different constraints call for different implementations, a fact that is not acknowledged by much of earlier work. For example, partial tiling provides benefits over full tiling for some problem sizes but not others, as evidenced by both our performance models and our experimental validation.

Our methodology is based on the tiling transformation, and our implementation targets HLS tools for their ability to implement efficient hardware into well-optimized high-level code. Targeting C/C++ instead of HDL code frees us from the need to implement many optimizations ourselves, by re-using those provided by the HLS tool. Our experience suggests that the use of domain-specific generative approaches is an effective strategy.

Finally, we have shown that high-level system modeling, using reasonably simple performance models based on a system-level view, was sufficient to predict performance with excellent accuracy. We believe that the overall approach of providing a few, carefully chosen design knobs and simple performance models to drive the exploration could be generalized to other classes of accelerators.

Chapter 6

Conclusion

6.1 Review of our Contributions

In this thesis, we present methodological and technical contributions to the design of hardware accelerators. As discussed in Chapter 1, such architectures are expected to meet strict resource or performance constraints. Often, these requirements can only be satisfied by implementing fine-grained trade-offs between hardware resources (e.g., power, bandwidth and area) and performance metrics (e.g., throughput, latency and accuracy). However, the design space is usually extremely large: identifying such trade-offs can be a challenging task. A core idea of this work is to tackle this complexity by providing analytical models to drive the exploration.

Performance-Accuracy Trade-Offs In the first half of this manuscript, we focus on *accuracy*. Trade-offs between accuracy and performance constitute a large field of opportunities for hardware designers. A classical example is the use of fixed-point arithmetic instead of floating-point to cut down hardware cost and reduce power consumption. Naturally, accuracy may not be reduced indefinitely: the implementation must satisfy application-specific accuracy constraints, which should be validated/enforced during design-space exploration.

Determining whether a given fixed-point configuration satisfies accuracy constraints happens to be a difficult problem in general. Two main classes of techniques are proposed in the literature: techniques based on simulations, and analytical techniques. Simulation is the most widely applicable, and is also very effective given enough real-world inputs. However, analytical models are much faster to evaluate than simulations, and can thus be used to explore a large number of design points in a short time, possibly identifying better solutions. The main challenge remains their limited applicability.

Prior to our work, analytical techniques could only handle one-dimensional systems. In Chapter 3, we extend earlier methods to multi-dimensional algorithms, such as image filters. We focus on *Linear, Shift-Invariant* filters, a generalization of 1D Linear Time-Invariant filters supported by other approaches. We propose a source-level design flow, where the architecture is specified as a C/C++ description. Our main challenges are thus:

- From a given a C/C++ algorithm, how can we retrieve a compact, mathematical representation of the underlying linear system ?
- From that representation, how can we derive a reliable accuracy model ?

The first challenge is addressed within the polyhedral model. We choose to represent LSI algorithms as Systems of Uniform Recurrence Equations (SURE). SUREs may also be seen as Multi-Dimensional Flow Graphs, a generalization of Signal-Flow Graphs used as an intermediate representation in earlier approaches. A major difference is that, contrary to SFGs, MDFGs/SUREs do not impose a "canonical" iteration order for each dimension. Complex, recursive image filters, that scan the image in all directions, may thus be represented. We use dataflow analysis to transform the program into a system of *affine* recurrence equations. Before this system can be recognized as SURE, some amount of transformation is required, including linearization/uniformization of dependences.

For the second problem – inferring accuracy models – we propose two different approaches. Both boil down to computing the integral and L^2 norm of the impulse response, but from dual points of view:

- In the *time-domain* approach, we derive these sums by unrolling / evaluating recurrence equations defining the system.
- In the *frequency-domain* approach, we use algebraic properties to derive the transfer functions of the system. We thus compute the sums above from the frequency response.

For the latter case, we propose a much simplified and more efficient version of the algorithm proposed by Menard et al [23]. Experiments show that our models are fast to derive. Their effectiveness is demonstrated on real-world input: results show an excellent match between measured and predicted accuracy degradations.

Finally, we show how the frequency-domain approach could be used, *before* WordLength Optimization, to handle the quantization of coefficients, a problem largely dismissed by other works.

Implementation Trade-Offs for Stencils on FPGAs In the second part of this thesis, we focus on implementation trade-offs for iterative stencil computations. Stencil computations form a widespread computational pattern used in many applications, from scientific simulations to embedded vision. Applications vary greatly in terms of constraints, domain size, dependencies and computational intensity.

At a high level, the performance of stencil implementations is mostly determined by computational throughput and memory performance. The tiling transformation, discussed in Chapter 4, is an essential tool to improve both aspects, by enhancing memory locality and enabling parallelism at multiple levels.

In Chapter 5, we present a systematic design methodology for implementing stencil computations on FPGAs. Our approach is based on a flexible design template, rooted in the tiling transformation, and featuring several design knobs:

- Maximum throughput can be controlled by adjusting the **unrolling factor** of the core datapath. This allows trade-offs between throughput and area.

- Larger **tile sizes** can be used to reduce bandwidth requirements at the cost of increased memory usage.
- The iteration space can be **tiled in only some dimensions** to further reduce bandwidth usage. The price is a partial loss of control over local memory size, as it is proportional to the size of the domain along untiled dimensions.

In addition, we provide simple performance models, derived from a high-level analysis, to serve as a basis for Design-Space Exploration.

To validate our design and models, we implemented our approach as a code-generation flow targeting Vivado SDSoC. For multiple performance targets, we identified interesting design points using our performance models. We then compared predicted values against measured performance/area. Our experiments show that our performance model is extremely accurate and that our resource usage model is accurate enough to identify the most interesting design points. Our models can thus serve as a basis for systematic design space exploration.

Finally, during this work, we realized the importance of memory contiguity to reduce latency and truly benefit from burst accesses on FPGA platforms. We thus devised a custom memory layout for our use case, currently only implemented for for the 2D-data stencils. However, it is still unclear whether this layout could be efficiently generalized to higher-dimensional stencils.

6.2 Perspectives

Our work opens several perspectives.

An obvious direction would be to extend our work on analytical accuracy to a wider class of programs, such as linear non-shift invariant algorithms or arbitrary non-linear filters made of "smooth" operations. A more interesting research direction might be to support non-polyhedral programs. Some of these algorithms, such as the Fast Fourier Transform, are regular, static control flow yet cannot be represented compactly with affine dependences. It is unclear how quantization noise propagates in such algorithms, and if unrolling can be avoided.

One could also imagine the use of analytical accuracy models in other contexts, for example to handle round-off errors in floating-point programs, or to predict the impact of transient ("soft") errors and approximate operators on the correctness of the program. A major difficulty is that, in such cases, errors do not verify the same statistical conditions as fixed-point quantization noise. For example, the latter two would probably show highly discontinuous error distributions and correlation between error sources.

Our work on stencils relies on a clear understanding of the major factors affecting the performance of stencil computations, as illustrated by the Roofline model. Similar observations also apply to many other algorithms. A more generic approach, targeting all algorithms, could probably be derived.

Properly extending our contiguity-optimizing memory layout beyond 2D stencils is another research direction. While this problematic is importance in practice to fully utilize the available bandwidth, it has not been largely investigated. One issue is to limit the number of additional memory buffers required to communicate tile

faces. Another one is to reduce the need for re-ordering memory. We suspect that contiguity and sequentiality are both desirable, but essentially incompatible properties: in higher-dimensional algorithms, one cannot have both in general. However, this question would deserve proper investigation.

Another idea would be to use tile-face based halo decomposition to reduce communication volume, for example by shrinking the wordlength of communicated data (thus introducing round-off errors at tile boundaries) or by applying compression at the tile-level.

Finally, the interaction of stencils and accuracy is unclear, and should be investigated. The ability of FPGAs to handle arbitrary wordlengths is one of their major selling-point, as significant reductions in area can be achieved when applications tolerate some level of round-off noise. Being able to properly characterize the impact of (for example) fixed-point encodings on the accuracy of stencils could allow for very effective trade-offs.

6.3 Conclusion

The main challenge of designing hardware accelerators is the huge design space of possible implementations for each application, especially when design dimensions such as accuracy are considered. Because different applications have different needs, a single solution cannot be expected to satisfy all constraints in every case. In this thesis, we defend a systematic, model-based approach to design. We have demonstrated the efficiency of this strategy in two different problematics (stencils and wordlength optimization). We believe that, as accelerators become more and more widespread, the use of domain-specific models will become essential to understand the implications of each design choice on the behavior of the system. Our work is a step in that direction.

Bibliography

[1] Article, "Accelerating fixed-point design for MB-OFDM UWB systems." `http://www.eetimes.com/document.asp?doc_id=1273357`.

[2] G. A. Constantinides and G. J. Woeginger, "The complexity of multiple wordlength assignment," *Appl. Math. Lett.*, vol. 15, no. 2, pp. 137–140, 2002.

[3] M. Nemani and F. N. Najm, "High-level area and power estimation for VLSI circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 697–713, 1999.

[4] C. Shi and R. W. Brodersen, "Automated fixed-point data-type optimization tool for signal processing and communication systems," in *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004* (S. Malik, L. Fix, and A. B. Kahng, eds.), pp. 478–483, ACM, 2004.

[5] D. Cachera and T. Risset, "Advances in bit width selection methodology," in *13th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2002), 17-19 July 2002, San Jose, CA, USA*, pp. 381–390, IEEE Computer Society, 2002.

[6] K. Parashar, D. Menard, R. Rocher, and O. Sentieys, "Shaping probability density function of quantization noise in fixed point systems," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, pp. 1675–1679, IEEE, 2010.

[7] L. P. Robichaud, "Signal flow graphs and applications," *TODO*, 1962.

[8] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow: Describing signal processing algorithm for parallel computation," in *COMPCON'87, Digest of Papers, Thirty-Second IEEE Computer Society International Conference, San Francisco, California, USA, February 23-27, 1987*, pp. 310–315, IEEE Computer Society, 1987.

[9] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, 1967.

[10] CAIRN, "ID.Fix." `http://idfix.gforge.inria.fr`.

[11] B. Widrow, I. Kollar, and M.-C. Liu, "Statistical theory of quantization," *IEEE Transactions on instrumentation and measurement*, vol. 45, no. 2, pp. 353–361, 1996.

[12] G. A. Constantinides, P. Y. Cheung, and W. Luk, "Truncation noise in fixed-point sfgs [digital filters]," *Electronics Letters*, vol. 35, no. 23, pp. 2012–2014, 1999.

[13] J.-M. Tourreilles, C. Nouet, and E. Martin, "A study on discrete wavelet transform implementation for a high level synthesis tool," in *Signal Processing Conference (EUSIPCO 1998), 9th European*, pp. 1–4, IEEE, 1998.

[14] G. A. Constantinides, P. Y. Cheung, and W. Luk, "The multiple wordlength paradigm," in *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pp. 51–60, IEEE, 2001.

[15] D. Menard, D. Chillet, F. Charot, and O. Sentieys, "Automatic floating-point to fixed-point conversion for dsp code generation," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 270–276, ACM, 2002.

[16] G. A. Constantinides, "Perturbation analysis for word-length optimization," in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pp. 81–90, IEEE, 2003.

[17] D. Menard, R. Rocher, P. Scalart, and O. Sentieys, "Automatic SQNR determination in non-linear and non-recursive fixed-point systems," in *Signal Processing Conference, 2004 12th European*, pp. 1349–1352, IEEE, 2004.

[18] R. Rocher, D. Menard, O. Sentieys, and P. Scalart, "Analytical accuracy evaluation of fixed-point systems," in *Signal Processing Conference, 2007 15th European*, pp. 999–1003, IEEE, 2007.

[19] G. Caffarena, J. A. López, A. F. Herrero, and C. Carreras, "SQNR estimation of non-linear fixed-point algorithms," in *18th European Signal Processing Conference, EUSIPCO 2010, Aalborg, Denmark, August 23-27, 2010*, pp. 522–526, IEEE, 2010.

[20] K. Parashar, *System-level approaches for fixed-point refinement of signal processing algorithms*. PhD thesis, Université Rennes 1, 2012.

[21] B. Barrois, K. Parashar, and O. Sentieys, "Leveraging power spectral density for scalable system-level accuracy evaluation," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pp. 750–755, EDA Consortium, 2016.

[22] G. A. Constantinides, P. Y. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 10, pp. 1432–1442, 2003.

[23] D. Menard, R. Rocher, and O. Sentieys, "Analytical fixed-point accuracy evaluation in linear time-invariant systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 55, no. 10, pp. 3197–3208, 2008.

[24] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, (Berlin, Heidelberg), pp. 283–303, Springer-Verlag, 2010.

[25] S. Rajopadhye, S. Gupta, and D.-G. Kim, "Alphabets: An extended polyhedral equational language," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 656–664, IEEE, 2011.

[26] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[27] V. Van Dongen and P. Quinton, "Uniformization of linear recurrence equations: a step toward the automatic synthesis of systolic arrays," in *Systolic Arrays, 1988., Proceedings of the International Conference on*, pp. 473–482, IEEE, 1988.

[28] W. Shang, E. Hodzic, and Z. Chen, "On uniformization of affine dependence algorithms," *IEEE Transactions on Computers*, vol. 45, no. 7, pp. 827–840, 1996.

[29] D. Barthou, P. Feautrier, and X. Redon, "On the equivalence of two systems of affine recurrence equations," in *European Conference on Parallel Processing*, pp. 309–313, Springer, 2002.

[30] K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens, "Verification of source code transformations by program equivalence checking," in *International Conference on Compiler Construction*, pp. 221–236, Springer, 2005.

[31] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the ACM*, vol. 14, pp. 563–590, July 1967.

[32] D. Monniaux, "An abstract monte-carlo method for the analysis of probabilistic programs," in *ACM SIGPLAN Notices*, vol. 36, pp. 93–101, ACM, 2001.

[33] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[34] M. Gardner, "Mathematical games: The fantastic combinations of john conway's new solitaire game "life"," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970.

[35] B. Chopard, "Cellular automata modeling of physical systems," in *Encyclopedia of Complexity and Systems Science*, pp. 865–892, Springer, 2009.

[36] G. B. Ermentrout and L. Edelstein-Keshet, "Cellular automata approaches to biological modeling," *Journal of theoretical Biology*, vol. 160, no. 1, pp. 97–133, 1993.

[37] S. Wolfram, "Cellular automata as models of complexity," *Nature*, vol. 311, no. 5985, pp. 419–424, 1984.

[38] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 311–320, ACM, 2012.

[39] C. Ancourt and F. Irigoin, "Scanning polyhedra with do loops," in *ACM Sigplan Notices*, vol. 26, pp. 39–50, ACM, 1991.

[40] P. Boulet and P. Feautrier, "Scanning polyhedra without do-loops," in *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pp. 4–11, IEEE, 1998.

[41] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th Symposium on Principles of Programming Languages*, POPL '88, pp. 319–329, 1988.

[42] D. Kim and S. Rajopadhye, "Efficient tiled loop generation: D-tiling," in *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, LCPC '09, 2009.

[43] S. Tavarageri, A. Hartono, M. Baskaran, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "Parametric tiling of affine loop nests," in *Proc. 15th Workshop on Compilers for Parallel Computers. Vienna, Austria*, 2010.

[44] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Dyntile: Parametric tiled loop generation for parallel execution on multicore processors," in *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, IPDPS '10, pp. 1–12, 2010.

[45] S. Shrestha, J. Manzano, A. Marquez, J. Feo, and G. R. Gao, "Jagged tiling for intra-tile parallelism and fine-grain multithreading," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 161–175, Springer, 2014.

[46] R. T. Mullapudi and U. Bondhugula, "Tiling for dynamic scheduling," 2014.

[47] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 28th Conference on Programming Language Design and Implementation*, PLDI '07, pp. 235–244, 2007.

[48] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pp. 1–11, IEEE, 2012.

[49] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," *SIAM Journal on Scientific Computing*, vol. 37, no. 4, pp. C439–C464, 2015.

[50] M. Korch, J. Kulbe, and C. Scholtes, "Diamond-like tiling schemes for efficient explicit euler on gpus," in *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pp. 259–266, IEEE, 2012.

[51] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for GPUs," in *Proceedings of the 2014 International Symposium on Code Generation and Optimization*, CGO '14, pp. 66:66–66:75, 2014.

[52] M. M. Strout, L. Carter, J. Ferrante, and B. Simon, "Schedule-independent storage mapping for loops," *ACM Sigplan Notices*, vol. 33, no. 11, pp. 24–33, 1998.

[53] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ACM SIGARCH Computer Architecture News*, vol. 19, pp. 63–74, ACM, 1991.

[54] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *ACM SIGPLAN Notices*, vol. 30, pp. 279–290, ACM, 1995.

[55] J. Cong, P. Li, B. Xiao, and P. Zhang, "An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers," in *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pp. 77:1–77:6, 2014.

[56] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Proceedings of the 3rd International Congress on Mathematical Software*, ICMS '10, pp. 299–302, Sept. 2010.

[57] M. Kunz, A. Ostrowski, and P. Zipf, "An FPGA-optimized architecture of horn and schunck optical flow algorithm for real-time applications," in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications*, FPL '14, pp. 1–4, Sept 2014.

[58] W. Luzhou, K. Sano, and S. Yamamoto, "Domain-specific language and compiler for stencil computation on FPGA-based systolic computational-memory array," in *Proceedings of the 8th International Symposium on Applied Reconfigurable Computing*, ARC '12, pp. 26–39, 2012.

[59] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, "A polyhedral model-based framework for dataflow implementation on FPGA devices of iterative stencil loops," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pp. 77:1–77:8, 2016.