DISSERTATION

AUTOMATING THE DERIVATION OF MEMORY ALLOCATIONS FOR ACCELERATION OF POLYHEDRAL PROGRAMS

Submitted by

Corentin Ferry

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2024

ABSTRACT

AUTOMATING THE DERIVATION OF MEMORY ALLOCATIONS FOR ACCELERATION OF POLYHEDRAL PROGRAMS

As processors compute power keeps increasing, so do their demands in memory accesses: some computations will require a higher bandwidth and exhibit regular memory access patterns, others will require a lower access latency and exhibit random access patterns. To cope with all demands, memory technologies are becoming diverse. It is then necessary to adapt both programs and hardware accelerators to the memory technology they use. Notably, memory access patterns and memory layouts have to be optimized. Manual optimization can be extremely tedious and does not scale to a large number of processors and memories, where automation becomes necessary.

In this Ph.D dissertation, we suggest several automated methods to derive data layouts from programs, notably for FPGA accelerators. We focus on getting the best throughput from high-latency, high-bandwidth memories and, for all kinds of memories, the lowest redundancy while preserving contiguity. To this effect, we introduce mathematical analyses to partition the data flow of a program with uniform and affine dependence patterns, propose memory layouts and automation techniques to get optimized FPGA accelerators.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Résumé en français

1.1 Introduction

Dans le domaine du calcul haute performance comme dans celui de l'informatique embarquée, on observe une demande croissante de performance de calcul combinée à une demande de réduction de la consommation énergétique des mêmes systèmes. Pour parvenir à ces fins, il est nécessaire d'utiliser des accélérateurs matériels et des architectures spécifiques.

L'accent mis sur le parallélisme au cours des années 2010 est à l'origine d'une importante proportion des gains en performance observés durant cette décennie. Celle-ci fut permise à la fois par la finesse de gravure des puces, qui permit de mettre davantage de ressources de calcul à disposition des utilisateurs, et par le développement de compilateurs auto-parallélisants, qui en permirent l'exploitation.

L'augmentation de la puissance de calcul se traduit par une pression accrue sur les systèmes mémoires qui, incapables d'y satisfaire, ralentissent alors les calculs. Ils sont aussi à l'origine d'une fraction importante de la consommation énergétique en raison des transferts inter-puces qu'ils effectuent. Pour faire face à ces deux défis, de nouvelles architectures mémoire sont en développement : certaines, comme les mémoires LPDDR, visent une basse consommation; d'autres au contraire cherchent à maximiser le débit en entrée et sortie comme les architectures HBM.

L'émergence de nouvelles architectures mémoire permet d'adapter le choix du sous-système mémoire au système développé. Ce seul choix ne suffit pas à garantir un débit effectif ni une consommation optimales. Afin d'exploiter au mieux l'architecture choisie, il est nécessaire d'optimiser le programme ou l'architecture de calcul afin d'optimiser son utilisation de la mémoire.

On peut regrouper les optimisations existantes en deux catégories : les optimisations de localité, qui visent à réduire la quantité d'entrée/sortie mémoire en exploitant les données présentes dans les caches et mémoires locales, et les optimisations d'accès, qui visent à réduire le temps de transfert total et ainsi obtenir le débit le plus élevé possible.

Dans cette thèse de doctorat, on s'intéresse particulièrement aux optimisations d'accès. Contrairement aux optimisations de localité, dont beaucoup sont aujourd'hui automatisées, les optimisations d'accès sont peu nombreuses, et demandent des modifications profondes dans la structure des données utilisées par le programme. De telles modifications peuvent permettre d'exhiber des motifs d'accès à la mémoire plus efficaces, car plus adaptés à la topologie des mémoires et au fonctionnement des contrôleurs. Notamment, les accès contigus, ou *burst*, sont les plus efficaces avec les contrôleurs DDR, mais aussi les plus difficiles à faire apparaître.

On développe dans cette thèse des méthodes d'analyse de programme permettant de connaître les besoins précis en données de chaque partie accélérée du programme, et de transformer la disposition des données ainsi que les motifs d'accès de l'accélérateur pour faire apparaître des accès contigus. Pour effectuer de telles modifications, rendant potentiellement inintelligibles les données à l'utilisateur, il est nécessaire de s'assurer de la correction des résultats fournis par le programme; cette assurance n'est possible que moyennant des hypothèses sur le comportement du programme, qui sont vérifiées à la compilation. Afin de permettre de transformer l'éventail le plus large possible de programmes, on cherche dans cette thèse quelles sont les hypothèses les plus faibles permettant de garantir la correction des transformations effectuées.

On propose ainsi dans cette thèse les contributions qui suivent :

- Une allocation de données contigue pour programmes utilisant un pavage polyédrique rectangulaire permettant aux accès contigus de traverser les frontières des tuiles,
- Une analyse à grain fin du flot de données du programme ainsi qu'un découpage de celui-ci en blocs de données pouvant être rendus contigus,

- Une transformation de programme utilisant l'analyse précédente pour compresser à la volée les données tout en assurant un volume d'entrée/sortie minimal,
- Une étude théorique sur les limites de validité des transformations proposées.

1.2 Allocation de données contigue pour pavage rectangulaire

Dans le chapitre 4, on propose une allocation de données spécifique aux accélérateurs FPGA visant à maximiser l'exploitation de la contiguité tout en contrôlant le volume d'accès dits *redondants*, c'est-à-dire le volume de données échangées bien que non utilisées afin de préserver la contiguité.

Lorsque le programme admet une représentation polyédrique, il peut être possible de lui appliquer un pavage (découpage de l'espace des itérations en tuiles de forme similaire), afin d'augmenter la localité des accès. Des données issues de l'exécution de chaque tuile sont nécessaires pour exécuter d'autres tuiles voisines; en outre, il est suffisant de n'allouer de mémoire que pour certains résultats intermédiaires. Lorsque les dépendances entre les calculs du programme sont dites uniformes, les données à allouer sont contenues dans les faces des tuiles voisines, dans une enveloppe rectangulaire de taille bornée et déterminable à la compilation. En choisissant de découper cette enveloppe en sous-enveloppes rectangulaires par projection, on obtient des ensembles de données qu'il est possible d'allouer en mémoire de manière contigue.

La connaissance du flot de données permet de connaître chacun des blocs à rechercher; on exploite alors le voisinage de ces blocs en mémoire, s'il existe, pour regrouper ensemble les accès aux différents blocs et obtenir les transactions les plus longues possibles. Ces deux niveaux de contiguité permettent d'obtenir une augmentation de la bande passante utilisée; les expériences qui ont été menées pour valider cette approche montrent qu'il est ainsi possible d'utiliser l'intégralité de la bande passante vers la mémoire de l'accélérateur FPGA.

1.3 Analyse et découpage du flot de données

Dans le chapitre 5, on s'intéresse au comportement mémoire des programmes pavés lorsque le pavage est composé d'hyperplans quelconques. On effectue pour cela une analyse plus précise du comportement des programmes admettant une représentation polyédrique. Chaque tuile a besoin de résultats intermédiaires calculés par d'autres tuiles voisines. Lorsque les dépendances entre les calculs sont *uniformes*, toutes les tuiles exhibent une empreinte d'accès similaire sur leurs voisines. On propose alors d'exploiter cette régularité et de découper les données transmises en groupes dotés de trois propriétés :

- Atomicité : La consommation d'un élément d'un groupe implique la consommation de l'ensemble du groupe;
- Unicité de l'origine (irredondance) : aucun élément ne peut faire partie de plusieurs groupes à la fois, et chaque groupe est originaire d'une unique tuile.
- Maximalité : Si deux éléments ont exactement les mêmes tuiles consommatrices, alors ils appartiennent au même groupe.

Ces trois propriétés caractérisent, et permettent de construire, les MARS (*Maximal Atomic irRedundant Sets*) qui peuvent être utilisés pour, outre la construction d'une allocation de mémoire, la détection et la correction des erreurs.

1.4 Allocation de données partitionnée et compression à la volée

Dans le chapitre 6, on étudie une application de l'analyse précédente afin de construire une allocation de mémoire qui permette d'avoir l'utilisation la plus efficace du bus mémoire tout en y faisant passer le volume de données le plus petit possible. Il est ainsi nécessaire d'avoir recours à la fois à la contiguité et à la compression des données.

Les propriétés des MARS permettent d'exhiber des blocs contigus, candidats à la compression en raison de leur propriété d'atomicité; afin de maximiser la contiguité, il est nécessaire de l'étendre à travers les blocs MARS ainsi créés. On formule un problème d'optimisation pour disposer en mémoire les MARS en cherchant à minimiser le nombre d'accès burst effectués. On propose également de compresser les données des MARS, et d'empaqueter les blocs compressés pour gagner à la fois en bande passante et en données. Cela permet de travailler avec des données dont le type et la largeur sont arbitraires, ce qui est courant sur FPGA, sans perte de bande passante.

On implémente un générateur de code permettant d'automatiser la création d'unités mémoires spécifiques, exploitant les MARS et leur compression, pour des accélérateurs FPGA.

1.5 Extension aux dépendances affines

Dans le chapitre 7, on s'intéresse aux limites des méthodes proposées aux deux chapitres précédents. Ceux-ci ne peuvent considérer que les dépendances entre les calculs lorsque celles-ci sont *uniformes*. Sont exclus les accès à la mémoire issus de dépendances non-uniformes, qui constituent pourtant l'essentiel des accès de certains programmes.

Le modèle polyédrique permet de représenter des dépendances affines, dont celles uniformes sont un cas particulier. Dans ce chapitre, on étudie ainsi la forme des dépendances affines qui peuvent engendrer des structures similaires aux MARS, tant sur des espaces de données d'entrée, que sur des espaces d'itérations pour le passage de résultats intermédiaires. Il est nécessaire que soit préservée l'uniformité des ensembles de données consommés par chaque tuile, c'est-à-dire la forme et la position de chaque ensemble, pour les qualifier de MARS.

Il ressort de cette étude que peuvent amener à des MARS les dépendances affines uniques, ou celles multiples partageant la même partie linéaire ou, à défaut, le même noyau. On propose également une méthode construction des MARS dans ces cas. Le cas des dépendances entre espaces d'itérations déjà tuilés est étudié, et une conjecture sur l'existence de MARS est établie. Cette conjecture soumet l'existence de MARS dans l'espace producteur, à l'existence d'un lien entre un déplacement dans l'espace consommateur et un déplacement dans l'espace producteur.

Chapter 2

Introduction

Performance optimization is a major topic of computer science today. The continued demand for performance drives the emergence of domain-specific hardware accelerators, under the form of fully application-specific integrated circuits (ASICs), accelerators implemented inside configurable chips (field-programmable gate arrays, FPGAs), or also as specialized units within general-purpose processors.

Creating domain-specific hardware is not sufficient to get a high performance; the hardware must be fully exploitable and exploited. To this aim, hardware and software developers have to respectively provide and uncover parallelism, and correctly exploit the memory hierarchy. While general-purpose processors feature caches, dedicated accelerators instead rely on carefully-tuned memory management as well as on-chip and off-chip data movement. In both cases, it is up to the hardware designers and application developers to make the best usage of bandwidth: they must seek to use every cycle to transfer useful data on the memory interface.

Compilers have largely improved over the 2010 decade in terms of both automatic parallelism extraction and cache utilization. In particular, they have control over the *memory access pattern*, which is the sequence of memory accesses performed by the accelerator. By leveraging value reutilization, they reduce the volume of data transiting through the memory hierarchy. This is not sufficient to get the best performance out of the memory subsystem. Designers and developers still have to manually perform *memory allocation*, that is, specifying where (physical chip, bank) and in which order (layout) the data used by their accelerator is stored. A good data layout has a significant impact on performance: it makes it possible to get contiguous, regular access patterns, leveraging the highest performance out of caches, prefetchers and controllers.

Memory architectures are also evolving and diversifying: high-performance computers are moving away from a central memory of a single type, to multiple memory chips close to the processing units, each with its own bandwidth and random access latency. The performance of a certain program depends on its utilization of the specific type of memory available and the memory architecture.

Tuning the data layout in memory has been known to improve performance of numerous programs, but remains a difficult problem to this day. Purpose-built layouts for specific applications exist, but automation is desirable. In this document, we propose generic memory layouts that improve bandwidth utilization, and automatically derive such layouts from the programs themselves using compiler analyses.

This chapter motivates and gives an overview of this Ph.D dissertation. Section 2.1 details and motivates the specific problems that are solved in this document; Section 2.2 gives an overview of the solutions proposed.

2.1 A need for further memory optimizations

In this section, we explain the continued necessity of optimizing memory accesses. While there is an immediate challenge caused by the continued increase in pressure on memory due to more parallelism being exploited, there is a long-term challenge posed by the widening of the space of memory architectures. New memory architectures will be under-utilized unless the data layout and access pattern are carefully tailored to them.

The next two subsections detail this argument; Sections 2.1.3 and 2.1.5 focus on the specific problems caused by data layout and access patterns.

2.1.1 An increasing pressure on memory due to parallelism

Processor chips keep containing more logic cells, mainly due to progresses in silicon manufacturing. Thanks to these area gains, more computation units can be put in the same silicon space. These units are designed to operate in parallel; in order to get the best performance from this silicon, a program needs to fully exploit its compute units, and must therefore *exhibit* parallelism.

New hardware is massively parallel

Current architectures leverage multiple levels of parallelism, from low-level vector operations to high-level parallel nodes in a machine cluster.

On CPU and GPU platforms, the parallel computation units are grouped in vector units, that can perform the same instruction on multiple operands. This model is called Single-Instruction, Multiple Data (SIMD). On FPGA and ASIC designs, parallel computation units are explicitly specified, and it is the job of the hardware developer to ensure that all processing units and memories can be actively used at all times. Large-scale systems feature many of these chips that all run in parallel, which adds another layer of parallelism to exploit, and more parallelism to be exhibited by the program.

Hardware designers also have optimizing compilers at their disposal called *high-level synthesis* (HLS) engines. These transform imperative code like that executed on CPU or GPU into a hardware architecture; in doing so, they also rely on parallelism extraction techniques. Section 3.3 covers some of their specific optimizations to produce parallel hardware. Deriving a parallel architecture and fully utilizing it are two interdependent challenges; there is active research on HLS tools with respect to scheduling and resource sharing ([44, 60, 82, 87]...).

Auto-parallelization increases memory pressure

Automatic extraction of parallelism has significantly evolved thanks to active and continued research for more than 30 years; such automation reduces the burden on the developer, who no longer has to explicitly state where the parallel computations can happen in the program. A series of *optimization passes* exist in this regard.

Optimizers and frameworks exist to leverage various levels of parallelism:

- Data-level parallelism: loop vectorization, which has been part of state-of-the-art compilers for years [6] and usually makes parallel computations in a loop explicitly appear.

Superword-level parallelism extraction [46] can group together isomorphic, parallel operations and issue a vector instruction for them.

- Instruction-level parallelism: a superscalar CPU can execute multiple instructions in parallel in its various execution units. Part of this parallelization task is done in out-of-order CPUs by the re-order buffers (ROB), provided they can find independent, non-conflicting instructions in the program. Compilers can also re-order instructions to extract ILP; this is especially used for in-order processors, like very-long instruction word (VLIW) processors ([59]).
- Thread-level parallelism: Loop tiling [41], on top of reducing memory accesses of a program by making it cache-friendly, gives coarse-grain parallel workloads implementable as parallel threads. Calls to OpenMP make it easy to exploit the parallelism leveraged by tiling.

The exploitation of each level of parallelism increases the memory traffic. To process more operands at a time, the processor needs to get these operands from its memory hierarchy, which must in turn handle all this extra traffic without becoming a bottleneck. When this happens, the design is said to be *memory-bound*.

2.1.2 A widening space of memory architectures

New workload-specific memory technologies

Memories have intrinsic characteristics: access latency, bandwidth and capacity. How each system is manufactured determines these three parameters, that are in a tradeoff: a high bandwidth requires parallel access to a large number of data, and therefore a partitioning of the data within the memory. Partitioning increases the random access latency due to additional arbitration and multiplexing. Higher capacities also increase the multiplexing needs, and in turn the latency.

Table 2.1: Benchmark figures of various memory technologies (DDR4, HBM) and platforms (U280 board, Amazon F1 datacenter platform).

Technology	Bandwidth	Latency	Capacity	Usage
U280 DDR4 [52]	38 GB/s	110 ns	32 GB	Main memory
Amazon F1 DDR [15]	57 GB/s	561 ns	Variable (GB)	Accelerator memory
U280 HBM [52]	612 GB/s	108 ns	8 GB	Accelerator memory

Each component in a system has its own bandwidth and latency requirements: for instance, CPUs tend to perform random accesses and need low access latency, while GPUs tend to perform regular access patterns and need a higher bandwidth. To cope with all these usages, a wide variety of memory technologies are simultaneously being developed. Table 2.1 shows benchmark results on various platforms using different memory technologies; latency, bandwidth and capacity can significantly vary from one platform and memory technology to another.

In order to get the best performance from these memory chips, how the programs use memory has to be tailored to the chosen technology.

More complex memory subsystems

Memory chips are generally part of a memory subsystem so that various components can access them. Central memory subsystems are among the most commonly implemented: a machine has a central memory all of its components (e.g., CPU, GPUs, daughter accelerator cards) can use, with a high latency, low bandwidth and high capacity. The pressure on such central memories and their limited bandwidth cause contention, hindering application performance.

Historically, caches and prefetchers have reduced the pressure on central memories; these features are private to each chip, and fail to address the contention issue where multiple accelerators exist, for instance in Systems-on-a-Chip (SoCs).

Distributed, networked memory subsystems are being developed to increase the overall bandwidth and relieve contention, in both small-scale systems (network on chip, NoC) and large-scale systems (software-controlled memory accesses).

On top of those challenges posed by individual memories, complex systems add a placement problem: accessing a random location in a complex subsystem has a high latency cost, even more so if the data is in a remote node of a distributed memory system. Adapting the placement of data within the subsystem becomes crucial to get performance.

2.1.3 Bandwidth-bound FPGA accelerators

On current architectures, the literature contains a significant number of references to bandwidth-boundness, particularly regarding FPGA accelerators. Two main memory subsystems surround FPGA accelerators today: DDR memories, shared with the rest of a System-on-a-Chip such as *Xilinx Zynq*; and the newer high-bandwidth memories, mostly available on high-end FPGA boards.

Cong et al. [19] evaluated DDR-based FPGAs and Graphics Processing Units (GPUs) by means of a benchmark suite called Rodinia, and bandwidth appeared to be a bottleneck on most tested benchmarks; an increase in available bandwidth to that promised by High-Bandwidth Memory (HBM), provided it is exploited, would lead to three FPGA benchmarks surpassing GPU at a better performance per watt ratio.

Some applications are intrinsically bandwidth-bound because they expose little reuse of their input data. This is the case of matrix-vector multiplication, used for instance in the fully connected layers of convolutional neural networks. Each element of the matrix is only used once, and therefore this application is known to be memory-bound on recent DDR-based FPGA platforms: in 2019, design space exploration for the Caffeine framework [86] to optimize the convolution part of a neural network on FPGA, resulted in the observation that all designs proposed by the framework would be bandwidth-bound. Acceleration of SpMV on FPGA is still an issue in 2023 [48], calling to a full utilization of the bandwidth by exploiting sparsity and data compression.

High-Bandwidth Memory is available in off-the-shelf FPGA boards such as *Xilinx Alveo U280*. While HBM increases the available bandwidth, it does not increase its utilization unless data is spread over all banks. HiSparse [26] is an instance of FPGA accelerator class for matrix-vector multiplication where such a partitioning is performed. Microbenchmarks [52] show that a single HBM bank can reach about 14 GB/s whereas multiple banks can give a 421 GB/s bandwidth. This underlines the need for an automated data layout and placement.

2.1.4 Recent improvements in the utilization of memory subsystems

When memory access is the bottleneck, all the available parallelism cannot be exploited. To relieve this, the program must be optimized with multiple respects:

- Improving the access locality, re-utilizing values already on chip and reducing memory accesses,
- Fully utilizing the memory subsystem when accesses are needed:
  - Avoiding cache misses: the memory accesses must re-use recently accessed addresses,
  - Using prefetchers: the sequence of accesses must be regular enough to be recognized by the prefetching mechanisms,
  - Exploiting the burst feature of memory buses: the addresses being accessed must be adjacent to each other.

Compiler passes have been developed to automatically address the above issues. This section gives a broad view of the optimizations they brought.

Improving the access locality

The first key to improving memory performance is to reduce the number of memory accesses, thereby improving the program's *arithmetic intensity* (AI) [78, 80]. Other ways to describe this in the literature are improving *locality* or reducing the *reuse distance*, with a common idea that the closer in time the multiple accesses to a value, the more likely it is to stay on chip registers or be present in a closer cache level.

Well-known locality optimizations *promote* memory to registers [53] and re-order instructions to re-use values freshly computed instead of sending them to memory. Value reutilization is a key component of major production compilers such as GCC or LLVM today. On FPGA accelerators, it is the developer's task to specify which data movements must take place as synthesis tools do not promote memory accesses to registers. The work of Wei et al. [77] is an instance of deep neural network where memory accesses are saved by keeping some intermediate results on chip. This approach involves partitioning the results with knowledge of the consumers of the intermediate results. One of the approaches in this Ph.D dissertation automatically computes such a partitioning and can be used for this purpose.

More advanced locality optimizations target caches. Loop tiling [41, 45] is one of them: it aims at breaking up a loop nest into atomic blocks that have a known memory footprint, typically the size of some level of cache. It is possibly applied at multiple levels [92], from registers [42, 43] all the way to parallel nodes. It is applicable and can be profitabke on CPUs as well as GPUs [36]. Although tiling can be applied purely as a loop transformation, it changes the execution order within the loop nest. Prior analysis, possibly automated [11, 12] must assert that it is legal, and profitable. Section 3.4.3 covers tiling in more detail using the polyhedral representation of a program. Tiling has made its way into production compilers; it is notably implemented in Polly [33] part of the LLVM optimizing compiler. Cache behavior is well-known, to the point it is possible to model and tune programs based on analytical *cache miss equations* [32].

Improving access patterns

Locality optimizations determine *which* addresses (containing which values) will be accessed at a certain point in time. Re-ordering these accesses can exhibit regularity in the address sequence and yield better performance.

Regular access patterns enable usage of prefetchers and contiguous access patterns (more restrictive than regular) enable the use of burst features, both improving performance. Prefetcher-friendliness optimizations notably try to expose regularity in the memory accesses that can be identified by the prefetcher.

Burst-friendliness is generally not an issue on CPUs and GPUs because the granularity of memory accesses is at least a cache line. Therefore, burst-friendliness is a problem mostly for FPGA and ASIC designs. Nevertheless, the fundamental issue at stake is not FPGA-specific: data contiguity is necessary to get good vector unit performance. Auto-vectorizing compiler passes such as superword-level parallelism (SLP) [46] extractors required such data to be contiguous, as data movement or shuffling within vector registers incurs an overhead that could lead to a slower code despite vectorization. More recent SLP extractors [49] can change the data layout if the accesses are regular enough, e.g. with a constant stride. It is also possible to re-order accesses within a DDR controller itself to exhibit bursts, and reduce the access latency seen from the outside of the controller [20].

The optimizations above do not guarantee the absence of cache misses, burst- or prefetcher-friendliness: how the data is laid out in memory can prevent the appearance of regular or contiguous access patterns. The next subsection deals with the specific issue of data layout.

2.1.5 Application-specific data layouts increase bandwidth utilization

A *raison d'être* of this Ph.D dissertation is that the choice of data layout in memory affects memory access performance. As said in Section 2.1.3, irregular and non-contiguous accesses prevent correct utilization of prefetchers, of auto-vectorizers but also cause high-latency scalar accesses on global memories.

There have been various attempts at modifying the memory layout that resulted in performance improvements. We give examples of them in the next subsections, and make the case that actually generalizing and automating them is a contribution towards fully utilizing current and future architectures.

Domain-specific layouts for FPGA accelerators

Due to the massive parallelism FPGAs offer, they are prone to memory-boundness. The literature contains a wide variety of methods to fully exploit the bandwidth of memories connected to FPGA accelerators. They mainly rely on increasing spatial locality by exhibiting data contiguity, which this dissertation largely covers. How this is achieved depends on the application, of which two examples are given below.

One way to exploit memory contiguity is to build streams of data, where the order of writing of the data is the same as the order of reading. It is possible thanks to works like SODA [13], to get such layouts for FPGA accelerators of applications like dense linear algebra or stencils.

An application that benefits from domain-specific layout is sparse matrix-vector product. On top of exploiting sparsity to compress the matrix data, further optimization is done with the layout of the compressed data, yielding a higher bandwidth utilization. It is possible, like done in HiSparse [26], to partition the data to match HBM banks, thereby exploiting all the banks of an HBM stack.

Fine-grain data layouts for vectorization

The works presented in this dissertation target domain-specific hardware accelerators. Yet, the problem being tackled is more general. It is for instance possible to improve the performance of instruction-set architectures in certain scenarios, such as vectorized programs, by using specific data layouts.

Vectorization is exploitation of a chip's vector units instead of equivalent scalar operations. It is achieved by using special instructions of the processor. Such instructions typically consist in replicating a single operation over multiple operands, ("single-instruction multiple-data", SIMD).

While vectorization helps in improving compute unit utilization, it requires careful data placement to limit the movement from, to and within vector registers. Solutions to this problem exist for domain-specific applications; for instance, for stencil computations, a specific data layout for the vector registers, distinct from that of the original arrays, can reduce the overall data movement from/to and within the vector registers [83].

2.1.6 The case for automation of memory layout optimizations

We make the case that memory layout optimizations should be made automatic, for the same reason as other performance optimizations. Such optimizations are hard to write by hand and, although nothing prevents the developer from applying manual optimizations, compiler-based ones provide:

- "speed": what would take a few hours or days for a human to design just takes seconds (e.g., manual RTL design is orders of magnitude more time-consuming than HLS).
- explorability: it is posssible to automatically generate a wide variety of designs, evaluate them and pick the best one.
- applicability: a single compiler pass can generate a memory allocation, possibly using target-dependent cost models but using the same optimizer code, and target-dependent backends generate all the data movement code necessary to use an allocation.

How to automate the derivation of a good memory layout is an issue in itself. Compilers can syntactically invert the dimensions of memory arrays using cost metrics [69]. Syntactic manipulations may however not be sufficient; Polyhedral optimizers can produce complex domains such as diamond tiles [9], and likewise complex access patterns, for which a complete re-engineering of the memory layout is needed to obtain good memory performance.

In this dissertation, we propose to use the polyhedral representation of programs to build memory layouts and propose algorithms and systematic methods to compute memory layouts. These techniques will work provided a certain number of hypotheses, determined for each technique, are verified.

2.2 Contributions of this Ph.D

We have made the case for the necessity of automated memory layout optimizations. Most of the contributions of this Ph.D. rely on one idea: like prior work generates parallel code from the polyhedral representation of a program, it is possible to generate contiguous memory layouts from the same polyhedral representation. Such contiguous layouts enable domain-specific hardware accelerators to have a high bandwidth utilization.

Chapter 3 covers the technical background all of this document relies upon; Chapters 4 through 7 cover the solutions proposed by this Ph.D, of which a brief summary is given below. They are described below, from the most restrictive scope of application to the least restrictive.

2.2.1 A multi-level contiguous memory layout for rectangular tiles

In Chapter 4, we focus on **programs tiled with a hyper-rectangular tile shape** and that have *uniform dependences.* For these applications, we can compute the *flow-in* and *flow-out* data as parts of thick faces, called "facets", of neighboring tiles. Therefore, we allocate data for these facets in such a way to enable contiguity within them and across them. These two levels of contiguity enable long burst accesses that result in a high bandwidth utilization, while keeping and a low redundancy. This allocation scheme is called Canonical Facet Allocation (CFA).

We propose an evaluation of this allocation scheme with an FPGA implementation of benchmark accelerators as task-level pipelines. For these, the external I/O phases (off-chip CFA from/to on-chip allocation) are generated using a compiler pass, and show that we are able to reach close to bus bandwidth.

2.2.2 An irredundant, atomic partitioning of inter-tile communications for arbitrary tile shapes

In Chapter 5, we consider **programs tiled with arbitrary shapes**, rather than rectangles as required by Chapter 4. Based on the observation that only a part of the flow-out of each tile is used by every of its consumer tiles, we proposes a **breakup of these flow-in and flow-out sets** that guarantees atomicity and irredundancy properties, and a method to construct such a breakup. The resulting sets, called Maximal Atomic irRedundant Sets (**MARS**), can be used in a variety of applications among which memory allocation, but also error detection.

2.2.3 A compressed and contiguous memory layout for arbitrary tile shapes

Chapter 6 is a use case of the MARS: it introduces **a memory allocation based on the MARS** that exploits its irredundant and atomic natures. These two properties allow for contiguity, data packing and compression, thus increasing the bandwidth utilization and decreasing the volume of data. This chapter brings in two contributions: through an optimization problem, it exploits the MARS irredundancy to minimize the number of input and output transactions for each tile; and, the atomicity property is exploited by automatically compressing and packing the data. A code generator is implemented and the resulting allocation and access pattern are evaluated on a selection of FPGA accelerators.

2.2.4 An extension of the data flow partitioning to affine dependences

One limitation of MARS is that it does not apply to affine dependences, that make up most of the input data on workloads such as matrix multiplication. Chapter 7 **extends the idea of atomic irredundant partitioning to data spaces with affine dependences**. Affine dependences bearing several properties (notably sharing the same null space) can yield such a decomposition; in this case, Chapter 7 proposes a method to partition the data into MARS. This chapter's theoretical contribution makes it possible to propose a data layout for the entire data flow with the same objectives as the one obtained in Chapter 6.

Chapter 3

Background

This dissertation relies on a fairly large stack of technical background: it uses both low-level hardware design techniques and high-level mathematical representations of programs. Between these, multiple levels of abstraction are involved and interact together. All of these levels of abstraction are necessary to efficiently use all the compute power of accelerators. This chapter aims at explaining the abstraction stack and giving design techniques related to memory accesses.

It first covers hardware memory architectures, their interfaces and controllers. It goes through the synthesis of compute and memory architectures from high-level code, then covers the high-level polyhedral abstractions used to analyze and transform the source program; finally, it explains how high-level transformations can have a controlled effect on low-level memory accesses.

3.1 Locality and performance

The performance of a program on a platform, i.e. how many computations it can perform per time unit, is determined by how the compute power of the platform and how the memory subsystem are utilized. In this dissertation, the focus is on the memory subsystem and locality optimizations; this section gives an insight into the effects of these optimizations on performance.

3.1.1 Roofline model

Quantiying the performance of a program is necessary to determine which kind of optimizations are needed, i.e. where there is room left for performance gains.

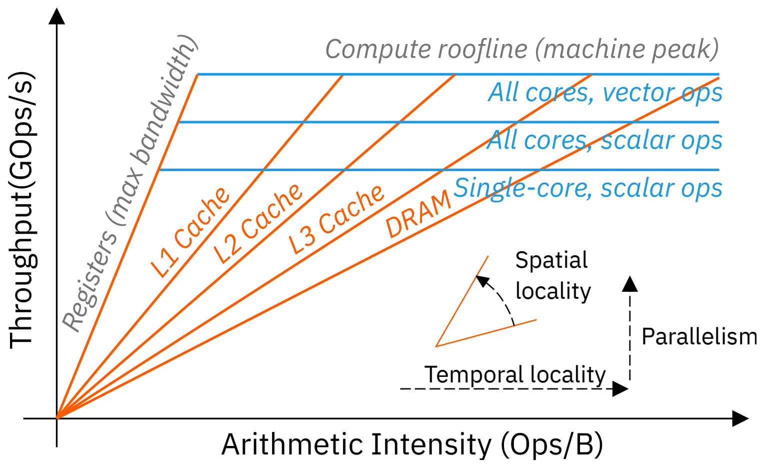


Figure 3.1: Roofline model

The *roofline model* [78, 80], illustrated on Figure 3.1 is a visual way to see the effects of locality improvements. The throughput a system can reach is bound when either a *memory roofline* or a *compute roofline* is hit. Compute rooflines represent the maximal throughput attainable with a given set of compute units; in the case of hardware designs, the maximum number of compute units that fit a given area budget are considered. Memory rooflines represent bandwidth caps intrinsic to the memory subsystem; there is one such roofline per level of the memory hierarchy.

To figure out the position of a program or hardware design in such a graph, one must compute its *arithmetic intensity* (AI), which is defined by:

$$AI = \frac{\textsf{number of arithmetic operations}}{\textsf{volume of I/O}}$$

A higher AI generally means a better achievable performance, thanks to a reduced likelihood of memory-boundness.

Memory-boundness results in an under-utilization of the on-chip parallel compute resources due to stalls; using a roofline graph, it is possible to explain how improving both spatial and temporal locality can relieve memory-boundness. The next subsection focuses on the two notions of spatial and temporal locality.

3.1.2 Access locality

The performance of memory accesses largely depends on how close together they are, in both time and space; correct utilization of the memory subsystem by improving the sequence of addresses in both time and space results in a good performance. There are two qualitiative notions of locality used to characterize the utilization of the memory subsystem by a program or accelerator.

Temporal locality characterizes how close in time the same addresses are being re-used by the program with a *re-use distance*. The lower this distance, the more likely the data is to be in a close level of cache; if the reuse distance is low, the data may still be available in on-chip registers. Improving temporal locality reduces the number of global memory accesses, and therefore increases arithmetic intensity. As shown in Figure 3.1, a higher temporal locality gives potential access to higher compute performance by limiting the pressure on memory.

Spatial locality, on the other hand, characterizes how close in space consecutive addresses being accessed are. If addresses are all adjacent to each other, the program has better locality than if they are not. For instance, a unit-strided access sequence has better spatial locality than a two-strided sequence, because the latter sequence does not exhibit any contiguity. In Figure 3.1, the effect of spatial locality on performance can be seen as the bandwidth utilization increases, thereby giving access to more compute power for the same arithmetic intensity.

3.2 Memory architectures and transfers

A memory architecture usually encompasses the storage locations of all *active* data with the exclusion of cold data stored on disks. Such memories are of the following type:

- Disk, usually for swapping. These have low throughput (hundreds of MB/s for mechanical disks to 32 GB/s for PCIe SSDs), very high access latency (milliseconds) and very high capacity (terabytes);

- DDR random access memory. This is where the bulk of a system's memory resides. It is attached to the main processor system or is on some daughter card, e.g. a GPU. It has high throughput (up to 128 GB/s), low latency (below 1 microsecond) and low capacity (below 1 TB usually).
- HBM random access memory. These have very high throughput (more than 500 GB/s) and a higher access latency than DDR (up to a microsecond) due to extra arbitration, while being usually smaller than regular DDRs. These are used for high-throughput processing.
- Caches, that are embedded on CPU or GPU chips. These have low access latency (nanoseconds) and low capacity due to the area they take and the expensive logic needed to manage them.
- Registers, that are located right next to the compute units. These have the lowest latency (ideally same-cycle access) but are also the rarest storage resource. They are the only type of memory onto which operations are directly performed, barring *compute-in-memory* engines.

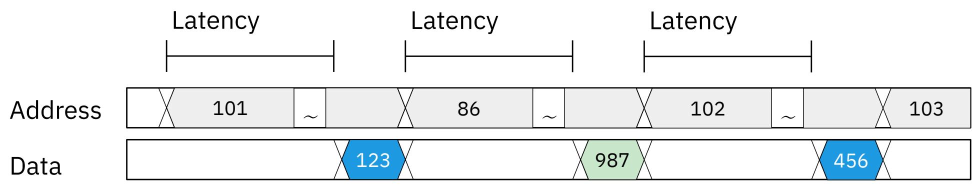
Registers are driven by the compute engines themselves, to which they belong. Other kinds of memories need controllers to be accessed.

3.2.1 High-bandwidth controller features

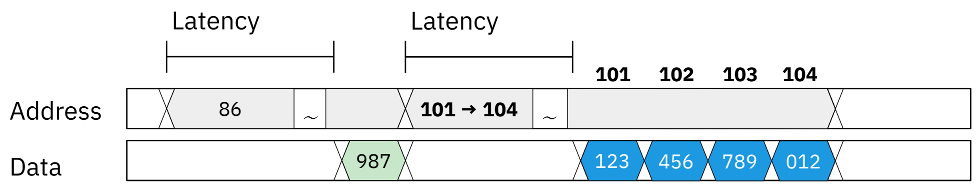
Memory controllers geared towards high bandwidth accesses feature ways to hide access latency of individual memory addresses, mainly by using some sort of pipelining. Burst-mode accesses and transaction-level pipelining are usually available; this subsection describes both features.

Burst-mode transactions

Memories are seldom dedicated to a single component, and are more usually connected to shared buses. Transaction-based shared buses require the use of two-way handshaking



(a) Scalar access pattern: there is a latency between each data transfer.



(b) Burst access pattern: multiple values are retrieved from consecutive addresses (101 to 104), without latency between each value. There is still a latency between accesses (non-contiguous in this example).

Figure 3.2: Burst accesses span over multiple addresses in a single transaction, amortizing the transaction latency over all elements accessed.

protocols before data transfer can begin. Arbitration and access protocols ensure that there are no access conflicts. During this time, the data transfer unit on the accelerator side is idle, as between data transfers in Figure 3.2a.

To amortize the access latency, shared buses usually feature burst-mode accesses, which perform several back-to-back accesses at consecutive addresses, as illustrated in Figure 3.2b. In order to use this burst mode, the accelerator must know:

- how much data it needs to transfer,
- which address is the first one to be accessed.

In first approximation, the latency of such accesses is modelled as an affine function: to transfer N_{words}, the number of cycles will be:

$$t_{\text{burst}}(N_{\text{words}}) = t_{\text{handshake}} + \frac{N_{\text{words}}}{\text{words per cycle}}$$

It is clear that the bigger N_{words} is, the lower the impact of $t_{\text{handshake}}$ and the higher the bandwidth utilization.

It is possible to hide the latency caused by the handshake protocol by overlapping actual data transfers with subsequent requests. Such overlapping can be compared to a two-stage access pipelining, splitting requests and transfers. This feature is present in the AXI4 bus, and exploited by some high-level synthesis tools described in the next section. Notably, when the end of a burst is contiguous to the start of the next burst, the tools generally can schedule them in a pipeline.

3.3 High-level synthesis

Generating hardware is usually done at the register-transfer level (RTL). This level of abstraction is very low and it is very difficult to reconstruct the algorithm from an RTL representation. This prevents rapid exploration of the design space, and requires skilled engineers to build hardware devices.

Well-known languages in the computer science field are mostly imperative languages such as C or C++. These languages do express the *what* of the program, i.e. which operations should happen on which operands, and abstract away the *how* which is bit manipulations. RTL on the other hand expresses the *how* and does not contain the *what*. When we want to design a piece of hardware, we start from the algorithm and go down to the bitwise operations that it needs to execute.

High-level synthesis (HLS) consists in automating hardware synthesis from an imperative language, and generating RTL from this language. Commercial tools such as Vitis HLS and Catapult accept C and C++ code as input, mostly due to these languages being used in embedded systems development and well-known from the EDA field.

3.3.1 From C/C++ to an architecture

Imperative languages specify an ordered sequence of instructions which lead to some result being computed. Control structures allow the program to repeat sub-sequences of instructions (loops) a finite number of times or until some condition is met, and also to selectively execute sub-sequences depending on conditions. It is possible using finite state machines to represent such programs that will execute the same sequence; the HLS engine will extract this representation; generating the corresponding RTL from a state machine is then a syntactic operation.

Doing this however leads to a purely sequential hardware accelerator with poor performance. It is necessary to express parallelism to get better performance.

C and C++ do not intrinsically express parallelism; it is difficult to figure out whether two operations can be executed in parallel or not, and which operations another one depends on before it can be executed. The developer, as well as some tools, knows these dependences. Special directives called *pragmas* are used to pass these hints to the HLS engine.

The core of an HLS engine does three main tasks to leverage RTL from imperative code: it creates the operators asked for by the user, schedules the operations, and binds each operation to an operator. While C and C++ contain the sequence of operations, scheduling and architectural directives exist to generate performant accelerators.

Architectural directives

Two kinds of parallelism can be expressed as compiler directives: synchronous parallelism (replication) and pipelining.

Replication, achieved by *unrolling* a loop, creates multiple copies of the same piece of hardware that run in parallel, as in Figure 3.3. For this operation to be profitable (i.e. result in an increase of throughput), consecutive loop iterations must not depend on one another; also, it increases resource usage because multiple hardware operators are needed to perform multiple operations in parallel.

Pipelining, shown in Figure 3.4, allows multiple consecutive iterations of a loop to be overlapped, by splitting each iteration into smaller operations called stages. Each stage executes an operation for a different loop iteration. Contrary to unrolling, pipelining does not require additional operators: it schedules a different loop iteration on each operator. For two iterations to be in the pipeline at the same time, they must be parallel, which means

```
for(int i = 1; i <= N-1; i++) {
#pragma HLS UNROLL
    A[i] = (B[i-1]+B[i]+B[i+1])/3;
}
```

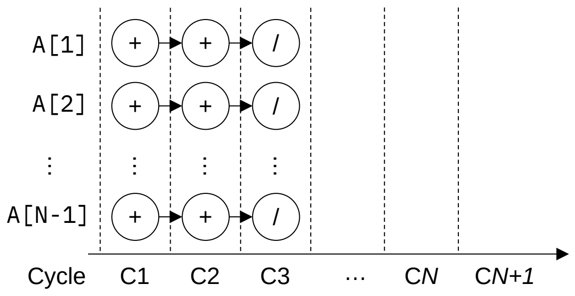


Figure 3.3: Loop unrolling in HLS creates parallel operators, and executes the loop iterations in parallel provided they are independent.

```
for(int i = 1; i <= N-1; i++) {
#pragma HLS PIPELINE II=1
    A[i] = (B[i-1]+B[i]+B[i+1])/3;
}
```

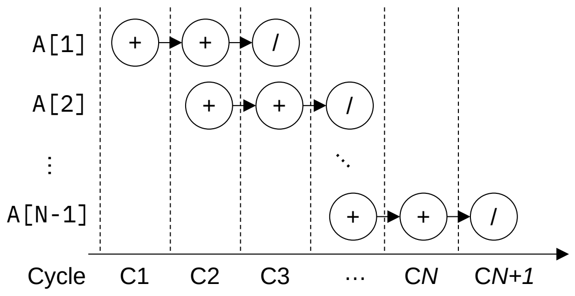


Figure 3.4: Loop pipelining in HLS overlaps multiple iterations of a loop, starting a new iteration every `II` cycles (here `II=1`), provided there is no dependence between them.

that the *reuse distance* (number of operations between the production of a value and its first use as an operand) is greater than the pipeline's depth.

It is possible to have multiple levels of pipeline, at different granularities. Fine-grain, data-level pipelines split operations on scalar elements to increase the processor's throughput. A coarse-grain, task-level pipeline is usually applied to high-throughput accelerators to multiplex input/output operations and computations, thereby hiding the I/O latency. Figure 3.5 shows such a "read-execute-write" pipeline, typically obtained with the following code:

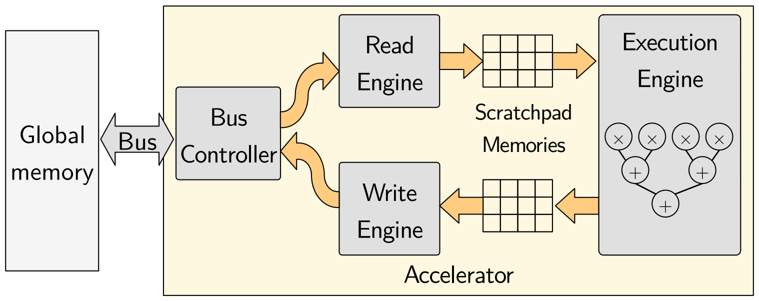


Figure 3.5: Task-level structure of an accelerator: read, execution and writeback happen separately.

```
void hls_toplevel(float* d_in, float* d_out, int i, int j) {
#pragma HLS INTERFACE m_axi port=d_in bundle=bus_ctl
#pragma HLS INTERFACE m_axi port=d_out bundle=bus_ctl
#pragma HLS DATAFLOW
    float input_arr[20][20];
    float output_arr[20][20];
    input(d_in, i, j, input_arr);
    process(input_arr, i, j, output_arr);
    output(d_out, i, j, output_arr);
}
```

Scheduling directives

An HLS engine is primarily used to create parallel architectures. However, even if multiple replicas of an operator are generated, they may not be used if the operations to be executed are indeed not parallel, i.e. they depend on each other. The HLS tool can, to some extent, recognize when there is such a dependence. In order to guarantee the correctness of resulting designs, an HLS tool is conservative and will assume dependences that do not exist; it will also fail to see dependences arising outside of the program, for instance due to side-effects or protocols.

The *DEPENDENCE* pragma can be used to indicate, within a loop, whether a variable depends or not on another one.

For instance, in the following loop, the array `A` is used in both read and write directions by the same statement, and the `DEPENDENCE` pragma indicates that no read-after-write dependence exists on this variable, i.e. that none of the read accesses to `A` depend on a write within this loop.

```
for(int i = 1; i <= HALF - 1; i = i + 1) {
#pragma HLS DEPENDENCE variable=A type=RAW inter false
    A[HALF + i] = (A[i - 1] + A[i] + A[i + 1]) / 3;
}
```

3.3.2 Memory accesses in HLS

The usual programming model for CPU architectures assumes a single virtual memory space into which all data is contained. A compiler then generates data movement code from memory to registers and vice-versa; the cache hierarchy is usually transparent and no code is necessary to manage caches.

FPGA architectures, like GPUs, contain scratchpad memories which management is explicitly specified by the developer: data movement in and out of local memories needs to be specified. Global memories, on the other hand, are accessed behind shared buses and the traditional memory model can be used on them.

The limitation of local memories is the number of ports (parallelism), whereas the limitation on global memories is the access latency.

Local memories

C and C++ do not have specific directives to specify the placement of data on local scratchpads and its distribution. An HLS tool like Vitis provides ways to automatically partition the C/C++ arrays into a finite number of scratchpads, which increases the number of available ports at the cost of a higher resource usage.

Global memories

Vitis HLS also provides global memory access directives to enable shared bus interfaces (such as AMBA, AXI); when such a directive is used, burst mode access (see Section 3.2.1) becomes available. Burst-mode accesses need to be supported by downstream bus controllers and memories. High-bandwidth controllers and chips, such as DDR and HBM chips, typically feature burst accesses.

Using burst mode is essential to get good access performance, as it amortizes the access latency of a single transaction over all data that is being accessed. In order to use it, the HLS engine must recognize a burst access, which is possible with one of the following constructs:

- A unit-strided loop with the counter being used as an address (e.g., a loop iterating over an array and copying an off-chip array into an on-chip buffer), and such that pipelining with an initiation interval (II) of 1 cycle (one new word each cycle) is possible:

```
for(int i = 0; i <= 9215; i = i + 1) {
  #pragma HLS PIPELINE II=1
  onChip[i] = offChip[5+i];
}
```

- A call to a system library (`memcpy()`):

```
memcpy(onChip, offChip + 5, 9215 * sizeof(float));
```

Note that bursts may be generated for loops where $II \geqslant 2$, but the throughput will be divided by II in this case.

Use of multiple layouts

High-performance accelerators must feature high-bandwidth accesses as well as high parallelism. It is therefore necessary to lay out the data on chip in such a way that parallel accesses are possible, while preserving contiguous accesses on the off-chip memory.

When such layout changes are implemented, they happen on the accelerator's side. Accesses to local memories and registers do not feature burst mode, and therefore do not need to

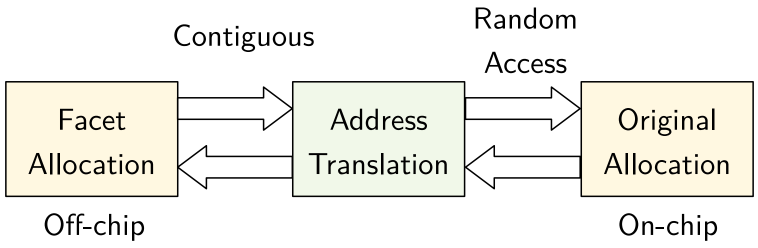


Figure 3.6: Changing layouts is necessary to get both contiguity and parallelism.

be contiguous. The HLS engine generates the routing and arbitration necessary to dispatch the data from a single global memory bus to on-chip distributed memories.

The layout separation and changes make it possible to decouple the search for off-chip and on-chip layouts.

3.4 Polyhedral model and tiling

High-level synthesis is done from code written in imperative languages. This abstraction can describe the exact sequence of computations and memory accesses to be peformed; yet, it fails to capture data flow information, which is the uses of every produced result, when every intermediate result is needed, which statement will use it, and when it will no longer be used and can be discarded.

Capturing data and control flow in a closed-form representation allows a compiler to work with programs as mathematical objects. One such representation is the polyhedral model, in which programs are represented with two kinds of mathematical objects: sets, on the one hand, and relations, on the other hand.

3.4.1 Abstractions

A polyhedral representation of a program comprises at least the following elements:

- An iteration space, which is a union of subsets of $\mathbf{Z}^d$, each point of which represents one instance of a program's statement (one specific valuation of the loop iterators),

- Data spaces, each of which is associated to a variable in the program. Each space is a subset of $\mathbf{Z}^n$, and each point in this space is a cell in the corresponding C-style array.

- A schedule, which is an affine function mapping each iteration to an integer or a vector of integers, specifying a relative order of execution of iterations,

- Array access informations (reads and writes) in the form of affine maps from the iteration spaces to the data spaces.

The polyhedral model requires sets of integer points and relations defined by affine constraints, which means that the relations between sets (memory access functions, dependence graph) are all affine. It specifies in what space and which order computations happen, but does not necessarily specify what is being computed. For this, it may be completed by equations, in which case it is a polyhedral equational model. An example of such a model is Alpha [47, 55].

We can consider two levels of polyhedral abstraction:

- The conventional polyhedral model, that is composed of a finite collection of statements (S), each having its own iteration space (D_S). Statement operations are abstracted away, except the memory accesses, i.e. reads ($\mathcal{R}_S$) and writes ($\mathcal{W}_S$) that take the form of affine maps. The program's variables (A), scalars or arrays, make up the data spaces D_A. For instance, the array `int A[200][100]` is mapped to the set $\{A[i,j] : 0 \leqslant i < 200 \text{ and } 0 \leqslant j < 100\}$. The statements reads and writes maps have their domains in D_S and their ranges in D_A. The model also comprises a schedule that is a partial order of the iteration space $D = \bigcup_S D_S$.

- The equational polyhedral model, that is composed of variables A, each defined over a domain D_A and which value at every point in D_A is given by an equation.

In this manuscript, we will only be considering the classical polyhedral model, that can be extracted from a sequential program using tools such as PET [75].

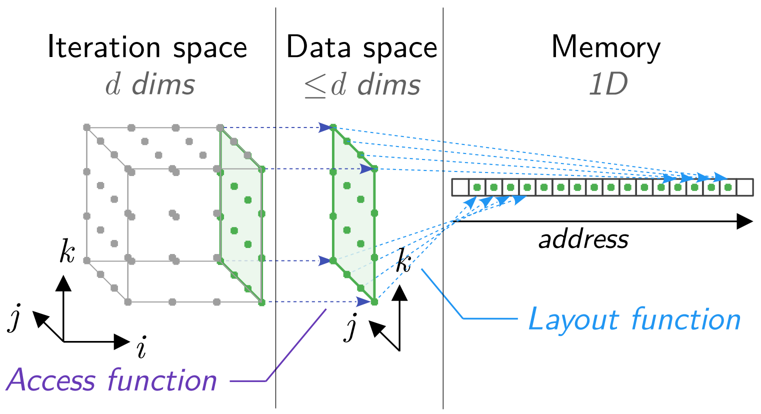


Figure 3.7: Iteration and data spaces in the classical polyhedral model. A memory map gives each data space point an address.

3.4.2 Modelling Dependences

In a C/C++ program, statements may have dependences between themselves, in the sense that one statement needs the result of a previous one as an operand. It is possible to model these dependences using affine relations; considering each instance of a statement has an iteration point in the iteration space, a depenedence relation will map the producer iteration point to the consumer iteration point.

Such dependence functions can be constructed algorithmically.

There are two families of dependences:

- "True" dependences, or flow dependences: these arise from the computation itself. They are intrinsic to the program. For instance, a recurrence equation $x_i = x_{i-1} + 2$ carries a dependence $(i-1) \mapsto i$. In a C/C++ program, these dependences are called *read-after-write*.

- Memory-based dependences: these arise from the memory allocation. When multiple iterations of a statement update the same memory cell, and the value contained at a given moment is used to compute another value, the source value must not be overwritten before that computation is done. These dependences are called *write-after-read*. Also, the order of writes into a cell may need to be preserved. This incurs *write-after-write* dependences.

Dependences prevent parallel executions of iterations: when an iteration, directly or indirectly, depends on another, these two iterations must be executed in sequence for the result to be correct.

Memory-based dependences are avoidable dependences: the memory allocation can be constructed independently of the computation itself. This dissertation largely relies on this fact and creates *fresh* memory allocations to favor memory bandwidth utilization.

True dependences, on the other hand, cannot be avoided. The polyhedral representation can however be transformed to find parallel iterations despite the existence of dependences.

3.4.3 Tiling

Tiling [41, 63, 68, 79] is a transformation of the polyhedral representation of a program that breaks up its iteration space into *tiles*, where each tile is atomic with respect to dependences. Such atomicity allows, for instance, synchronization-free execution of independent parallel tiles on different processors.

Tiling is implemented in most state-of-the art polyhedral compilers [11, 12, 33]. This subsection explains the principle of tiling, gives an optimization problem to tile for performance, and explains the consequences of tiling on the program's data flow.

Principle

Tiling is a program transformation that seeks to increase parallelism, locality and reuse. It consists in splitting an iteration space in similarly-shaped blocks, each of which containing a certain number of iterations. Such blocks must be able to be executed atomically, in such a way that multiple tiles may be executed in parallel.

Loop tiling increases both access locality and reuse. It can be applied at multiple levels, for instance to leverage thread-level and instruction-level parallelism at the same time.

Tiling for thread-level parallelism is done so that the footprint of each tile fits in a local memory dedicated to this thread. It may be the L1 or L2 cache in a CPU, or a scratchpad memory in a GPU.

Legality Tiling must respect causality between iterations; therefore, it is not always legal. Because tiles are atomic, there must in particular be no dependences in the PRDG that would cross a tile boundary both ways, as in Figure 3.8.

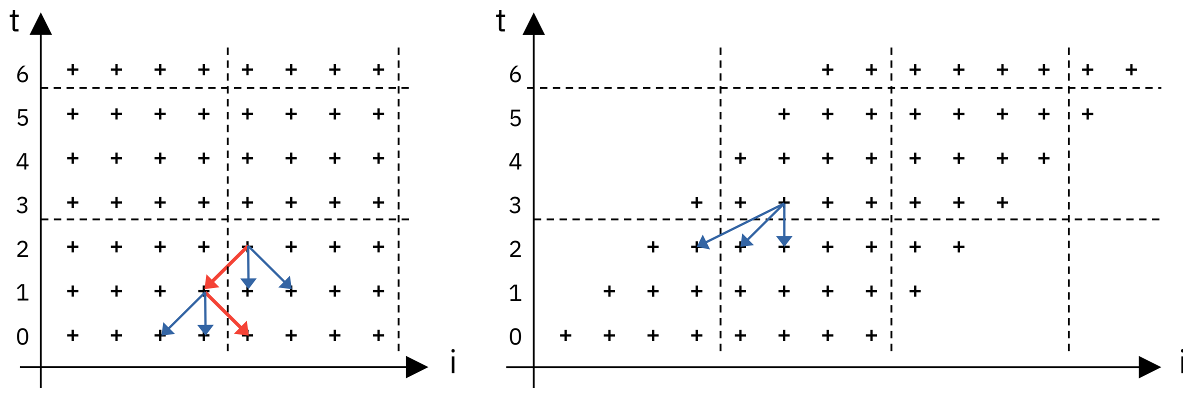


Figure 3.8: Rectangular tiling (cutting with a family of hyperplanes $\mathcal{H}_i : i = 4k, k \in \mathbf{N}$ and $\mathcal{H}_t : t = 3p, p \in \mathbf{N}$) is illegal on the left, and becomes legal on the right after time skewing.

Mathematically, tiling cuts the iteration space into two subspaces, each one on one side of an hyperplane. In a two-dimensional space, a hyperplane is a line; in a three-dimensional space, a hyperplane is a plane. Tiling legality is checked by projecting the *dependence vectors* onto a vector perpendicular to the *tiling hyperplane.* All the dependencies must be in the same direction with respect to that vector (which means that they all cross the tiling hyperplane the same way, or don't cross it at all).

A tiling hyperplane $\mathcal{H}$ is legal if, for all dependencies $\vec{b_i}$ the following property is true. Let $\vec{c}$ be a vector orthogonal to $\mathcal{H}$ (by definition of a hyperplane, there is such a vector). Then:

$$\begin{cases} \forall i, \vec{b_i} \cdot \vec{c} \leqslant 0 \\ \text{or} \\ \forall i, \vec{b_i} \cdot \vec{c} \geqslant 0 \end{cases}$$

This means that we can "cut" the space into two along the hyperplane $\mathcal{H}$ and all dependencies that cross the hyperplane, do so in the *same direction.*

Multiple tilings can be applied to the same iteration space: in a two-dimensional iteration space, it is possible to obtain two-dimensional tiles, for instance those in Figure 3.8.

Rectangular tiling

Rectangular tiling is the simplest kind of tiling: each tile is an hyper-rectangular block within the iteration space. The families of tiling hyperplanes are all parallel to one canonic axis. Figure 3.8 shows an instance of rectangular tiling with a Jacobi-like dependence pattern.

Rectangular tiling does not require a polyhedral representation; it can be syntactically performed as the composition of two loop transformations, loop strip-mining and loop interchange. For instance, in Figure 3.9, the loops over `i` and `j` have been tiled with tile size 4, resulting into outer loops over `ii` and `jj` with a step of 4 and outer loops over `i` and `j`.

```
int i, j;
for(i=1; i<25; ++i) {
  for(j=0; j<33; ++j) {
    B[i][j] = A[i-1][j]
      + A[i][j] + A[i+1][j];
  }
}
```

```
int ii, jj, i, j;
for(ii=1; ii<25; ii+=4) {
  for(jj=0; jj<33; jj+=4) {
    for(i=0; i<4; ++i) {
      for(j=0; j<4; ++j) {
        B[ii+i][jj+j] = A[ii+i
  -1][jj+j]
          + A[ii+i][jj+j] + A[ii+i
  +1][jj+j];
      }
    }
  }
}
```

Figure 3.9: Rectangular tiling applied to two dimensions of a loop nest.

Other kinds of tiling such as diamond tiling [9] exist, with different stances on the parallelism / locality tradeoff.

Tiling for Parallelism and Locality

Impact of tile size on performance Common wisdom with tile size selection says that it is done in such a way that some level of cache or some local memory is filled with data. This is not always true, however. The issue of tile size selection is well-studied, e.g. [16]

or [56]. This chapter provides techniques where we assume good tile sizes can be found through exploration.

The data space size also has an influence on performance: tiling, especially with powers of two as tile sizes, may result in poor performance due to a large amount of cache misses, despite the tile fitting in the cache. This phenomenon, called cache set conflict, is due to the cache associativity, and can be avoided by padding the data arrays [37, 65].

Automatic selection of tile shape and size Tiling hyperplanes and sizes can be algorithmically tuned to enhance parallelism and locality. PLuTo [11, 12] is one of the most used tiling algorithms, that maximizes parallelism across tiles and locality inside the tile; it will select tiling hyperplanes such that the number of dependences that cross each hyperplane is minimal.

Data Flow in the polyhedral representation

A tile is a collection of iteration points, i.e. instances of statements that require input values and produce output values. These values may come from other statement instances, be used by other statement instances, or be program input or output. For individual statement instances (individual iterations), these inputs and outputs may be both internal to the tile, (i.e. the producing iterations are inside the tile) and external. In the latter case, external production or consumption means that the results need to be communicated, for instance through global memory.

We call *flow-in* and *flow-out* iterations those that, respectively, require values from iterations external to the tile and produce values consumed by iterations external to the tile. The *flow-in set* and *flow-out set* (as shown in Figure 3.10), proper to each tile, are the collections of, respectively, all flow-in and flow-out iterations.

3.4.4 Notations

Throughout this document, the notations and conventions of Table 3.1 will be used.

Table 3.1: Notations used throughout this document

Symbol	Description	Remarks
S	Statement	Imperative language statement in the source program.
D_S	Statement iteration domain	Subset of $\mathbf{Z}^d$, also called *iteration space* for that statement.
d	Iteration space dimensionality	Number of surrounding loops for a statement.
$\vec{e_i}$	Basis vector	$\vec{e_1}, \ldots, \vec{e_d}$ is the canonical basis of D.
D	Iteration domain	Union of iteration domains of all statements.
$\vec{x}$	Iteration	An instance of a statement for a specific valuation of its iterators: $\vec{x} = (x_1, \ldots, x_d)$. In particular, $\vec{x} \in D$.
$\mathcal{A}$	Variable	Symbol used in the source program.
$D_{\mathcal{A}}$	Data space of variable $\mathcal{A}$	Subset of $\mathbf{Z}^d$, where d is the number of dimensions of $\mathcal{A}$.
$\mathcal{R}_S$	Reads of statement S	$\mathcal{R}_S : D_S \to \bigcup_{\mathcal{A} \in \mathsf{Variables}} D_{\mathcal{A}}$
$\mathcal{W}_S$	Writes of statement S	$\mathcal{W}_S : D_S \to \bigcup_{\mathcal{A} \in \mathsf{Variables}} D_{\mathcal{A}}$
$\mathsf{Aff}(D, E)$	Affine functions $D \to E$	Denotes all affine functions from space D to space E.
$\mathsf{ker}(A)$	Null space	Set of points which image by A (linear part of an affine function) is the null vector, $\vec{0}$.
$\mathbf{B}$	Dependences	Set of dependence relations. Also called *dependence polyhedron* in the literature.
Q	Number of dependences	
$\vec{b}$	Uniform dependence vector	Notation used when a dependence function is a translation (uniform). Then, $\vec{b} \in \mathbf{B}$.
$B(\vec{x}) = A\vec{x} + \vec{b}$	Dependence function	Notation used when a dependence function is affine. Notably, $B \in \mathbf{B}$.
H	Tiling hyperplane	Defined by its standard equation $\sum_{i=1}^{d} a_i x_i = 0$.
$\vec{n}$	Normal vector	Unit vector normal to hyperplane H.
$\vec{\mathbf{n}}$	Scaled normal vector	Vector normal to hyperplane H, that translates by one tile in the space.
t	Number of tiling hyperplanes	
$\vec{t}$	Tile coordinates	$\vec{t} = (i_1, \ldots, i_t)$.
$T(\vec{t})$	Tile	Defined by $\{\vec{x} \in D : \forall i \in [\![1; t]\!] : t_i s_i \leqslant \vec{x} \cdot \vec{n}_i < (1 + t_i) s_i\}$
$\mathcal{T}$	Tile space	Set of all $\vec{t}$s.
$\overline{\mathcal{T}}$	Interior tiles	Tiles that have no intersection with the iteration space's boundaries.

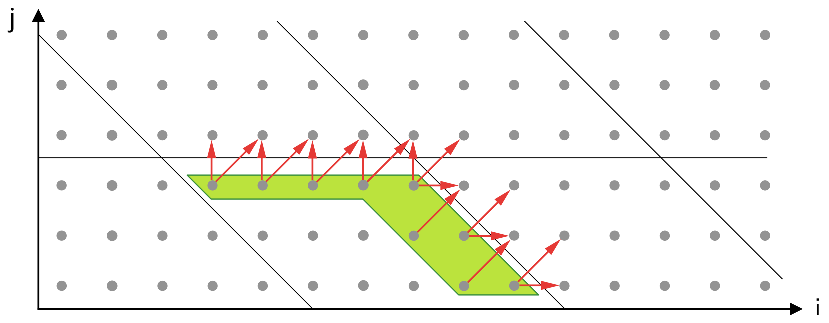


Figure 3.10: Flow-out set of a tile with a Smith-Waterman dependence pattern. Dependences pointing to other tiles are shown producer to consumer:.

Table 3.2: Polyhedral transformations needed to obtain a given HLS target

HLS target	Polyhedral transformation
Task-level pipeline	Coarse-grain loop tiling
Parallel loop iterations	Parallel scheduling change of basis
Pipeline initiation interval	ADA & scheduling minimizing reuse distance
Memory acccess pattern	Data space change of basis

3.5 Polyhedral compilation and accelerator synthesis

The toolkit used in this dissertation comprises distant abstractions, from high-level polyhedral abstractions to low-level synthesis techniques.

The tools used to bridge these abstractions are called *polyhedral compilers*. They are optimizing compilers, that extract and process a polyhedral model from an imperative program, and are usually source-to-source as their output is imperative programs. High-level synthesis tools take their output and map it into low-level, synthesizable RTL. It is then up to FPGA synthesis tools to issue a programmable FPGA bistream from this RTL.

3.5.1 Extracting parallelism within the polyhedral model

A polyhedral representation contains all memory access information of the program, and can capture dataflow information. This information can be used to detect possible parallelism within a program.

Extracting dataflow information from a polyhedral representation is possible for instance using Feautrier's array dataflow analysis [27]. This information is carried in the form of affine relations:

$$\mathbf{B} = \left\{ B : (\vec{x} \mapsto A\vec{x} + \vec{b}) \in \mathsf{Aff}(D_{S_1}, D_{S_2}) : \vec{x}_1 \in D_{S_1} \text{ is consumed by } \vec{x}_2 \in D_{S_2} \Leftrightarrow \vec{x}_1 = A\vec{x}_2 + \vec{b} \right\}$$

Otherwise said, an iteration called *consumer* is dependent on another one, the *producer*, if and only if the consumer iteration is the image of the producer iteration by an affine dependence relation.

Tiling can be applied to D so long as the dependences in $\mathbf{B}$ are not violated. Two kinds of tiling are envisioned:

- Coarse-grain tiling, for node-level parallelism. The choice of tiling hyperplanes and tile sizes are driven by the available resources on the node, for instance the cache size of the CPU or the amount of Block RAM available on the FPGA.
- Fine-grain tiling, for data-level parallelism. The choice of tiling hyperplanes is driven by the amount of usable parallelism as well as the local memory allocation and constraints: one wants to maximize the utilized parallelism while avoiding port contention (on FPGA/ASIC) or excessive register pressure (on ISAs).

Searching for a suitable tile size and shape is an optimization problem. There are domain-specific solvers, such as PLuTo [8, 11, 12] that computes, via an optimization problem, tiling hyperplanes that maximize the tile-level parallelism, and minimize the inter-tile dependences (and therefore the inter-tile communication volume).

3.5.2 From a parallel polyhedral representation to a parallel architecture

Once a parallel schedule is obtained from the polyhedral repsentation, we create an architecture in two steps: polyhedral code generation, and optimizing high-level synthesis tools.

Manually optimizing an HLS design at the source level is a tedious process; domain-specific source-to-source compilers [18, 50, 51, 82] automatically optimize existing HLS designs, e.g. with respect to parallelism and locality. Some of these compilers such as PolyOpt/HLS [60] and POLSCA [89] are targeted at HLS engines, and use polyhedral optimization techniques. These involve the same two steps as defined below

Polyhedral code generation

The polyhedral representation of the program contains information on which instances of which statement must be executed (iteration space), and in which order (schedule). Assuming the schedule is legal (i.e. the causality conditions are met and there are no memory conflicts), it is possible to generate a program that describes the iteration space in the selected schedule and correctly computes the desired result.

Code generators such as CLooG [3] are used to generate such codes in the form of loop nests.

There are two technical challenges HLS polyhedral code generation. First, programs are generated in an imperative, sequential language, making parallelism implicit whereas a parallel architecture is desired. Second, multiple distinct regions of code need to be generated when loop tiling is applied: while the accelerator itself executes the interior of a tile, the schedule of the tiles is managed outside the accelerator and must be separately generated.

Parallelism is implicitly contained within the schedule, using the following idea: if two points in the iteration space have the same schedule, then they *may* be executed in parallel. Likewise, if an entire subspace of the iteration space has the same schedule, this entire subspace is parallel.

Polyhedral code generators issue sequential code even for parallel regions of an iteration space; a sequential loop that describes a parallel subspace is still such that all its iterations can legally be executed in parallel. Two annotations are needed to pass the parallelism down to the HLS engine: an *unrolling* directive, that will expose all iterations of the loop (the code generator itself may unroll the loop if the HLS engine is not able to do so), and

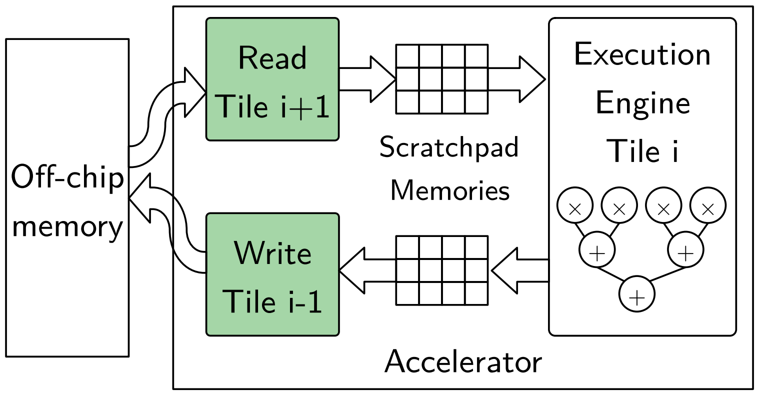


Figure 3.11: Macro-pipeline structure: read-execute-write. Our contribution focuses on the read and write stages.

no-dependence directives to make it explicit to the HLS engine that the loop is indeed parallel without requiring subsequent analysis from it.

Not all iterations that *may* be executed in parallel must be scheduled in parallel: one can specify a cap on the number of parallel execution units to be generated. This number is set using the unrolling factor.

Optimizing high-level synthesis

Loop tiling naturally yields a "read-execute-writeback" macro-pipeline structure as illustrated in Figure 3.11: because tiles can be executed atomically, all I/O operations can happen before and after execution. HLS tools such as Vitis HLS support such macro-pipelines through manual code annotation, but through a restricted set of conditions of the pipeline (i.e. absence of cyclic dependency between stages).

In this dissertation, we take advantage of the separation of stages to generate distinct I/O functions that do not affect the execution stages. We only seek to optimize transfer times and memory bandwidth usage. Existing tools [14, 60, 66, 71] can already optimize the execute stage and take advantage of massive operation-level parallelism (thanks to loop pipelining and unrolling).

High-level synthesis engines do not use the polyhedral representation, and have a limited dependence analysis ability. Hints provided at the polyhedral code generation time can supersede its analysis and are important to enable parallelism.

3.5.3 From a polyhedral representation to a memory access pattern

The core contributions of this Ph.D rely on automatic computation of memory layouts and access patterns from the program's polyhedral representation. Implementing these layouts and access patterns involves generating data structures and access code from the polyhedral representation. The generated code must match specific patterns to obtain burst accesses, needed for a high bandwidth utilization.

Data spaces to access code

Data spaces in the polyhedral model are represented as multi-dimensional sets of integer points. These sets have a natural *lexicographic order*, given by the coordinates of the points they contain. Enumerating all points according to that lexicographic order gives a layout of the data in memory.

That enumeration is performed by the same code generation tools as the iteration spaces, such as CLooG [3]. I/O functions are generally generated from maps between two spaces, the on-chip memory and off-chip memory spaces. Such code copies data between memories where it may have different layouts. For instance, the following code stores a transposed version of a matrix on chip:

```
for(int i = 0; i < 200; i++) {
  for(int j = 0; j < 200; j++) {
    local_array[i][j] = ext_array[j + OFFSET][i + OFFSET];
  }
}
```

Burst inference

HLS tools can infer burst memory accesses depending on the target interface. In the case of a shared bus (e.g. AXI, PCIe), which is commonly found for off-chip accesses, a *burst* access may occur if the bus supports it and the compiler recognizes access to a series of consecutive addresses. In burst mode, no cycle is spent stalling for a new value after a one-shot initialization latency, which yields full utilization of the available bandwidth.

Tools such as Vitis HLS 2022.2 exploit this using with either a call to a HLS-specific `memcpy` routine, or through some form or pattern matching in the source code. To obtain a burst access from a loop nest, generated by a polyhedral code generator, it is usually sufficient that the code verifies the following:

- The loop bounds are statically known, constant integers,
- The loop counter is incremented by one,
- The memory accesses use the loop counter as an index variable.

Examples of codes from which a burst can be inferred are given in Section 3.3.2.

Chapter 4

A Multi-level Contiguous Data Allocation for Rectangular Tiles

4.1 Introduction

The performance of accelerators is driven by two main factors: parallelism and memory accesses. To get the best performance, computations must not be slowed down by memory accesses; this throughput constraint is even more prevalent on an FPGA accelerator, where memory transfers and computations happen in parallel, and the slowest of memory and computations determines the actual throughput. As seen in Chapter 2 and recalled in Figure 4.1, both temporal locality and spatial locality, i.e. the amount of memory accesses and the bandwidth utilization must be improved to get the best performance.

Temporal locality is achievable using loop tiling, as seen in Chapter 3. Tiling caters to both locality (i.e. bounding the resources) and transfer parallelism: because tiles are atomically executed, it is possible to schedule all transfers before and after the computations.

To be applicable, tiling often requires a pre-transformation of the iteration space (e.g. skewing), that makes it legal to apply tiling. A skewing transformation changes the memory access pattern, at the expense of spatial locality: even rectangular tiles may no longer have rectangular footprints on the data arrays. However, to fully exploit high-bandwidth-memories, one needs to transfer large contiguous chunks of memory. We therefore suggest to create a memory allocation from the tiled version of the program, with the objective to maximize the data access contiguity.

In this chapter, we create a data allocation from the tiled version of a program, after any other legality transformations have been applied. We allocate data in a contiguous manner

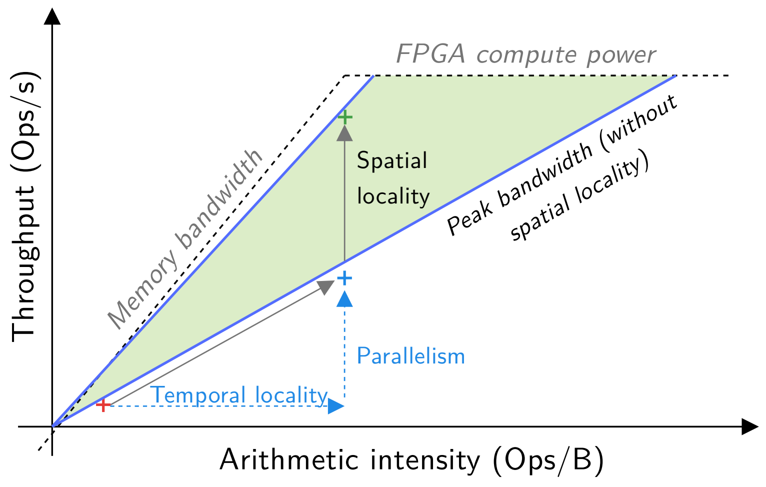


Figure 4.1: The method proposed in this work improves spatial locality, giving access to higher performance due to fewer memory-induced stalls.

inside the data produced by each tile, but also across neighboring tiles, thereby maximizing the utilization of the memory bandwidth.

The contributions of this chapter are:

- A memory layout and access pattern with two levels of contiguity for programs tiled using rectangular tiles,
- A proof-of-concept compiler pass that automatically applies this memory layout,
- An evaluation of the performance of this layout, that shows it can fully utilize the available bandwidth, while keeping the accelerator area about the same as other layouts.

This chapter is organized as follows: Section 4.2 describes the context of this work and existing ways to increase bandwidth utilization; then, Section 4.3 describes the construction of our burst-friendly off-chip layout. Finally, Section 4.5 provides an evaluation and comparison with the state of the art.

4.2 Related work

This work is primarily a compiler optimization technique for FPGA accelerators, and in that sense, it belongs to, and is complemented by, a large variety of automated optimizations. These optimizations target the computation schedule (e.g., to improve temporal locality), the memory access pattern (e.g., cache conflict elimination, burst extraction with existing allocations) the architecture (e.g. parallelism extraction, local memory partitioning). In this section, we mention some optimizations that need to be applied along with the global memory access optimization this work proposes, and also cover other domain-specific memory layout techniques.

4.2.1 Improving temporal locality

The first optimizations to apply that targets memory utilization improve temporal locality, and reduce the number of accesses.

Temporal locality increases the arithmetic intensity, by reducing the volume of data transferred per arithmetic operation. One way to obtain this result is to re-use the data already locally present on-chip or in a close level of the memory hierarchy. The following methods increase the arithmetic intensity of a given program.

Modifying the program's schedule to favor data reuse reduces the need for off-chip transfers. Loop tiling [41, 79] is the main technique in this aim. The primary target of tiling was to reduce the cache miss rate of CPUs, that have a cache hierarchy, but the technique also applies to software-programmable accelerators such as GPUs [36], and thanks to high-level synthesis tools, it also applies to application-specific hardware accelerators as well [60].

Sharing on-chip resources to maximize their usage: On hardware accelerators, fine-tuning of on-chip memory allocation may eliminate off-chip accesses. Memory cells may be allocated multiple times, shared across tasks that do not interfere [77], increasing the usefulness of each memory cell, thereby reducing the amount of off-chip traffic.

4.2.2 Increasing the effective bandwidth

The root of the memory bandwidth issue is found in the massive parallelism the hardware can provide. When it is fully used, the memory latency may be higher than the compute latency, due to how the memory subsystem is designed and used. In this case, the design is said to be *memory-bound*. Such a limitation may be caused, for instance, by port contention, cache conflict misses, scalar accesses to global memory. All of these issues find their root in a sub-optimal memory access pattern, a sub-optimal physical placement (layout) of the data, or both.

Program and data transformations of several kinds can be used to adapt the access pattern to the data layout, or adapt the data layout itself to the program; proper use of the memory and its access interfaces result in an increase of the memory bandwidth to a value closer to the nominal memory chip bandwidth. Further increase of the *effective bandwidth*, which is the amount of useful data transferred per unit of time, may be achieved using data compression techniques.

Our work is positioned among these techniques, but it is also essential that temporal locality and on-chip performance optimizations are applied to exhibit sufficient parallelism and create the demand for bandwidth.

Optimizing memory access pattern and layout

Sub-optimal usage of memory bandwidth causes a drop down in effective bandwidth. It may be due to where the data is located in memory and the schedule of access requests. The following methods can increase the effective bandwidth by playing with the data layout, the access pattern, or both.

Eliminating access conflicts Access conflicts may occur in all memory architecture, when one wants to use a physical port or memory cell to convey multiple data at the same time. Mapping the data to other cells or addresses will resolve the conflicts. On FPGA chips, parallel memory access patterns may incur port contention. Such a phenomenon is common with on-chip memories, but may happen on multiple-port memory architectures such as high-bandwidth memory (HBM). Bank partitioning [17] is a key and widely used technique to bank conflict prevention, and is available in commercial high-level synthesis tools. Conflicts also happen in set-associative caches when multiple addresses share the same set and therefore the same physical memory cells in the cache. Appropriate array padding [37] reduces such conflict misses at the price of an increase in off-chip array size.

Exhibiting burst accesses by re-scheduling memory accesses It may be possible to exhibit burst accesses by changing the access order of a given set of data addresses. If communications and computations are not separate [5], then the execution's schedule is also the memory access schedule; a loop transformation that maximizes DRAM row and burst use is found. Such a process changing the loop's execution order, it may break temporal locality. When communications and computations are executed separately and on-chip memories are used as scratchpads [60], the global memory access pattern does not need to match the on-chip access pattern. In this case, it is possible to improve memory bandwidth usage while keeping the same on-chip performance.

Re-allocating data in a burst-friendly layout Regardless of the data layout, it is always possible to make burst accesses to memory, however, the proportion of useful data accessed by such bursts may be low. It is possible to change the data layout to exhibit a high-usefulness burst access pattern. Data tiling [45] is one such technique; in the specific case of dense matrix product (GEMM), block matrix layout or *data tiling* as illustrated in Figure 4.2 (c) is known to have excellent spatial locality [35]. The dependence pattern of GEMM is such that the footprint of a tile can exactly fit a data tile, making it ideal layout for such an application. There is a trade-off between size and shape of data tiles and the usefulness of the burst accesses they permit, explored in works such as [58].

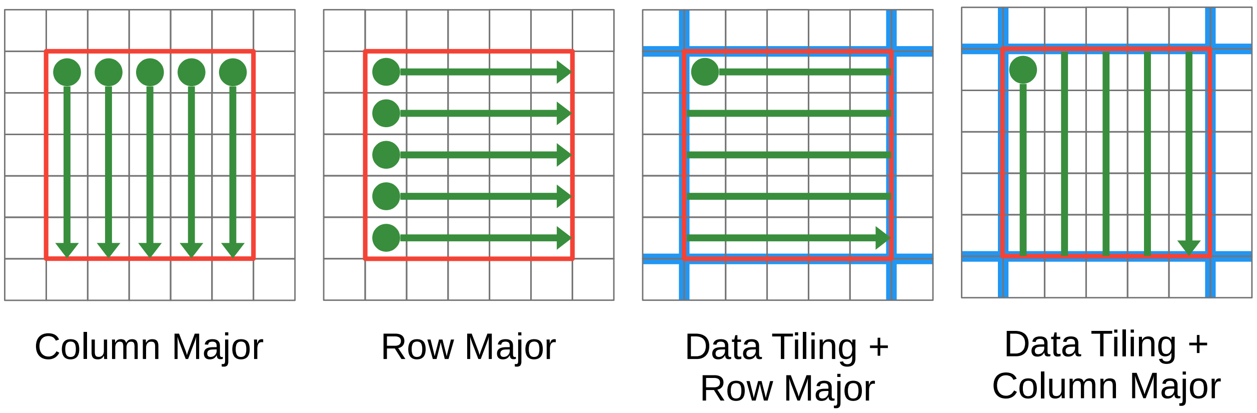


Figure 4.2: Memory layouts (non-tiled and tiled) for a 2-dimensional data space, and access patterns for a tile. Each dot is the start of a burst access, that spans until the arrow tip. Data tiling layouts enable a single burst access per tile instead of one per row/column.

Increasing effective bandwidth by compressing data

Another class of solutions around the bandwidth issue is using compression, whether lossy or lossless. Ozturk et al. [58] created a dynamic lossless compression engine that acts like a cache, where local data is compressed before being sent out to memory, and decompressed when coming from memory. Compression combines two advantages: it saves both memory and bandwidth, at the cost of using extra cores or on-chip area to perform it. A major pitfall of compression combined with data tiling is that it requires to read or write a full tile even to access a single point from it.

Lossy compression enhances throughput as well, at the price of errors. Maier et al.'s perforation [54] is a form of lossy compression, as is the well-known JPEG algorithm. Nakahara et al. [57] use it on CNN inputs; their method is to compress the CNN input on the host using the lossy JPEG algorithm, decompress it on FPGA chip, and perform the inference there. As expected, accuracy goes down as the JPEG quality factor decreases, and there is a trade-off with the obtained speedup.

Sun et al. [72], tackle the same bandwidth problem as ours by using a mix of compression and data layout. Although this approach does not rely on polyhedral dependency analysis, it features the same base idea: group together data that is being used together.

4.2.3 Automatic synthesis of optimized hardware

This work is an automated memory layout optimization for FPGA accelerators, that needs to be accompanied by more FPGA-specific optimizations. There is a longtime sustained community interest in automating optimization of hardware design, including high-level polyhedral optimizations (as opposed to low-level RTL tuning). This has resulted in compilers targeting FPGAs being developed, and we give some of them below.

Early work on polyhedral compilation was already targeting hardware design; Le Verge, Mauras and Quinton in 1991 [76] were using the Alpha language to automatically derive a systolic VHDL circuit from an Alpha polyhedral specification.

More recent work on HLS tools made it possible to explore tiling options for FPGA or ASIC accelerators (Pouchet et al., 2013 [60]) on various performance metrics such as latency, area, power consumption, memory bandwidth. Polyhedral transformations are done using specific tools, such as Pluto (Bondhugula et al., 2008 [10]) that automatically identifies tilable loop nests and applies tiling.

The most recent advances in tools bear a focus on data movement and memory issues. The SODA framework [13] automatically generates a dataflow-like pipeline structure with FIFO-ordered off-chip accesses; the data is automatically transformed have a specific allocation for this to work. This approach turns the data layout into a specific predetermined layout, independent of the actual dependence pattern, whereas ours finds a data layout and an access pattern for each accelerator in function of the dependence pattern.

Xiang et al., 2022 [81] propose an approach that is complementary to ours, with an HLS code generator that infers the entire data movement and memory allocation, both off-chip and on-chip. This approach does not rely on fine-grain dependency analysis, and therefore has to keep the inner layout of each array; however, the location of each array in memory is carefully chosen so as to minimize the latency and maximize the bandwidth. Our approach is complementary: we make use of fine-grain dependency analysis to transform the inside of the arrays.

4.3 Canonical Facet Allocation

One of the two main contributions of this chapter is a memory allocation technique based on the polyhedral representation of the program. It is based on the observation of Deest et al. [24, 25], that when rectangular tiling is applied, the flow-in and flow-out data is contained in the faces of each tile. Provided the intermediate results of a tile stay on-chip throughout the entire execution of a tile, it is therefore only necessary to clearly identify these faces and provide a memory allocation for their data, that can then be made contiguous.

Our idea is to create such a contiguous memory allocation and add extra contiguity opportunities, by having faces from neighboring tiles adjacent in memory. This enables contiguous accesses across faces, hence further contiguity.

In this section, we explain how the data layout and corresponding access functions are built. Section 4.4 explains the compiler flow and code generation phases to implement the allocation described here into an accelerator.

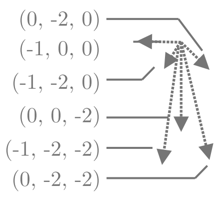
4.3.1 Description of the method

To create a Canonical Facet Allocation (CFA), we start from a program tiled with rectangular tiles. We first need to determine which data to take, and this must include all of the *flow-in* data. Using the observation that this data is contained into faces of tiles adjacent to the one being executed, we create multiple data spaces, one per face, and create *projections* that map iteration results to the newly created spaces.

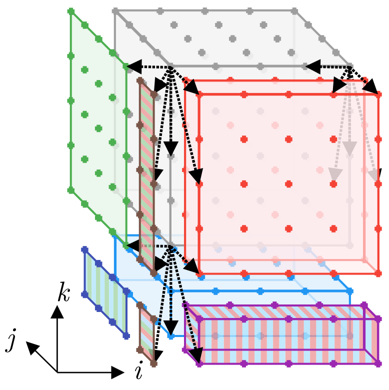
Once the data spaces are created, their layout has to be determined. For CFA, the chosen layout is contiguity at multiple levels: the data produced by a tile must be written to bulk, contiguous spaces to preserve its contiguity (*intra-tile contiguity*), and it must be possible to make contiguous accesses spanning several regions (*inter-tile contiguity*).

To meet these two levels of contiguity, two techniques are applied:

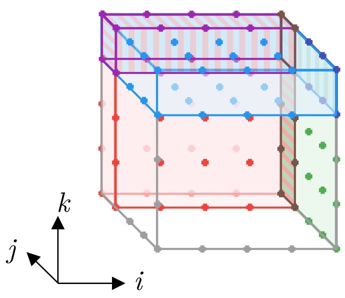
- data tiling,
- array dimension permutation.



(a) Dependence pattern



(b) Iteration-wise flow-in set, split per producer tile



(c) Iteration-wise flow-out set (facets)

Figure 4.3: An instance of flow-in and flow-out sets. The flow-out set is the union of thicker versions of the tile faces (facets), while the flow-in set is composed of an union of either whole or partial facets. Some flow-in sets are adjacent in the iteration space; CFA reflects this adjacency in memory.

The next subsections give a more formal view of how the projections (Sec. 4.3.4), the data tiling scheme (Sec. 4.3.5) are created, and how to pick the array dimensions to be swapped to enable multi-level contiguity (Sec. 4.3.6).

4.3.2 Hypotheses

In this chapter, we make the following assumptions on the programs to be transformed.

Uniform dependences: It is, in general, impossible to guarantee that the flow-in and flow-out of each tile are bounded. It is however sufficient that all the dependences are uniform (i.e. they are translations) to have this guarantee. We will therefore make this hypothesis, and can write the dependences as a set of vectors:

$$\mathbf{B} = \left\{ \vec{b}_i : i = 1, \ldots, Q \right\}$$

Also, this hypothesis does not imply that all memory accesses are uniform - this only applies to accesses to read-write arrays (those that hold intermediate results).

Rectangular tiling: We assume tiles are rectangular in all dimensions; using polyhedral tools, it is notably possible to change the iteration space basis so that rectangular tiling becomes legal. Therefore, before applying CFA, we expect such a pre-processing to have been done if necessary.

Non-sparse data: It does not make sense to apply CFA on highly sparse data. The data layout of CFA is dense per construction, so that uniform dependencies yield uniform memory accesses. Using it with sparse data would lead to a significant amount of avoidable redundant data transfers. Sparse data should use a sparse representation instead.

4.3.3 Definitions

We use the following terminology in the next subsections:

Projection: We use a restricted definition of projection, considering only orthogonal projections. They are used to strip one dimension from a multi-dimensional space. For instance, the projection p_k such that

$$p_k(i_0, i_1, \ldots, i_k, \ldots, i_d) = (i_0, i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_d)$$

removes the k-th dimension from an d-dimensional space.

Tile: A tile is an hyperrectangle, subset of the iteration space. Its size is $s_1 \times \ldots s_d$ in general; in the 3-dimensional example of Figure 4.3, tile size is respectively s_1, s_2, s_3 where s_1 corresponds to tile size on the i axis, s_2 for the j axis and s_3 for the k axis. A tile has coordinates $i_1, \ldots, i_d$ along each axis; in the example, these are i_1, i_2 and i_3 along respectively the i, j and k axes.

Facet: A facet is an hyperrectangular set of iterations that is contained in the flow-out set of a tile, and that contains at least one face of the tile.

First-level neighbor: A first-level neighbor of a tile is a neighbor that is reached with a move along a single canonical axis. For instance, $\vec{t_1} = (i_1, i_2 + 1, i_3)$ is a first-level neighbor of $\vec{t} = (i_1, i_2, i_3)$, but $\vec{t_2} = (i_1 + 1, i_2 + 1, i_3)$ is not.

Second-level neighbor: A second-level neighbor of a tile is a neighbor such that is reached with a move along exactly two distinct canonical axes. For instance,

$\vec{t_1} = (i_1 - 1, i_2 + 1, i_3)$ is a second-level neighbor of $\vec{t} = (i_1, i_2, i_3)$, but $\vec{t_2} = (i_1 + 1, i_2 + 1, i_3 + 1)$ and $\vec{t_3} = (i_1 + 1, i_2, i_3)$ are not.

k-th level neighbor: By extension of the above two definitions, a k-th level neighbor of a tile is a neighboring tile that is reached with a move along exactly k canonical axes.

In this chapter, we assume **all dimensions have been tiled**. We will use the following notations, as per Table 3.1:

- $E \subset \text{vect}\,(\vec{e_1}, \ldots, \vec{e_d})$: d-dimensional iteration space
- $\vec{b_1}, \ldots, \vec{b_p}$: dependence vectors, such that rectangular tiling is legal (see Section 3.4.3). We assume all dependence vectors are backwards in all dimensions: $\forall i, j : \vec{b_i} \cdot \vec{e_j} \leqslant 0$.
- $N_1 \times \cdots \times N_d$: iteration space size
- $t_1, \ldots, t_d \in \mathbf{N}^*$: tile sizes (note $t = d$ because all dimensions are tiled)

4.3.4 Multi-projection: contiguity along multiple directions

The first aspect of Canonical Facet Allocation is to determine which data should be part of facets transmitted across tiles. We show that making multiple projections of an iteration tile gives all the *flow-out* data. Moreover, each projection goes to a different data space, each having its own layout and access pattern. Each layout will then be tailored for contiguity. This section explains the multiple projections and their rationale.

Rationale

In a tile, not every iteration produces a result that is needed in other tiles. Therefore, only the result of select iterations, called the *flow-out set* of a tile, goes to memory. Visually, in Figure 4.3c, the flow-out set is made out of three "slabs" (facets), each of which is consumed by different tiles (see Figure 4.3b). Each of the "slabs" is consumed by the tile it is immediately adjacent to, and partially by other tiles. Therefore, each slab should be a contiguous piece of memory.

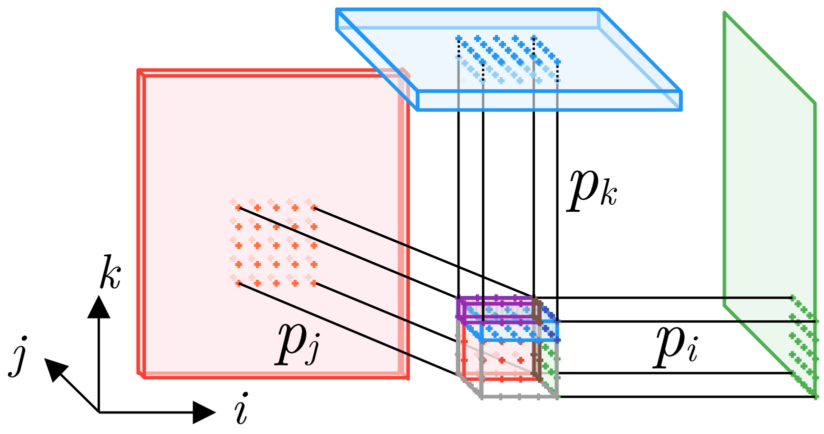


Figure 4.4: Multi-projection: each facet (see Figure 4.3c) is projected (mapped) to a data space. Choice of contiguity comes later on.

We map iterations to memory using a *projection* from the iteration space to an array with fewer dimensions. To get the best spatial locality, this array should mirror the neighborhood relations in the iteration space: the values produced by two neighboring iterations in *any* direction should also be neighbors in memory. Given that memory is a one-dimensional space, only one such neighboring direction in the iterations can be mapped to memory. This is not sufficient to map each "slab" to a contiguous piece of memory.

Multi-projection, illustrated in Figure 4.4 separates the "slabs" by cutting the flow-out set into three (partially overlapping) pieces. Each piece of the cut is mapped to a distinct array (data space). In Figure 4.4, there is a data space for each of the canonical hyperplanes (i, j), (i, k) and (j, k), to which iterations are mapped with resp. projections p_k, p_j, p_i.

There are as many directions of contiguity as projections. Having more directions of contiguity decreases the number of read transactions, at the expense of a higher storage redundancy, and a higher writeback time.

Intuitively, with uniform dependences, the flow-in set of each tile is parallel to the tile's faces. It therefore makes sense to have one projection per face of the tile.

The next two paragraphs state how each projected data space is created, first with an example, then in the general case.

Construction example

We start with a visual way to construct the data spaces, from a 3-dimensional iteration space, using Figure 4.3. The idea to construct the projected data space is:

1. To determine how thick every facet is, and

2. To create a function that maps iteration coordinates to data array coordinates.

In Figure 4.3c, the part of the flow-out parallel to hyperplane (i, j), in light blue, has a thickness of 2: consumer tiles will need the result of the two uppermost (i, j) planes ($k \in \{3, 4\}$). The dual way to see it is the flow-in (Figure 4.3b): when moving the dependence pattern along the bottom plane of a consumer tile, the iterations this plane depends on (part of which the blue slab below the consumer tile) are located two planes below it.

The idea is to make data spaces that are as thick as the dependence pattern "plunges" into the neighboring tiles. For each projection, it is the maximum length of every dependence vector along the normal vector to the hyperplane we are projecting on. In the example of Figure 4.3, the dependence pattern is in Figure 4.3a. The thickness of data space for hyperplane (j, k) is the maximum absolute value of the component along the i axis of every dependence vector. We determine it to be 1: only the rightmost (j, k) plane of iterations is needed, and is mapped to a two-dimensional array (`Dtype facet_i[`N_j`][`N_k`]`). We name it `facet` because it will ultimately contain data tiles holding entire facets.

The mapping from iteration coordinates to array coordinates for this example (see Figure 4.4) is the following:

$$P_i(i, j, k) = (j, k)$$

and its domain is the rightmost face of each tile:

$$D(P_i) = \{(i, j, k) : i \equiv 4 \mod 5\}$$

Note that, although we do not do it at this point, the above projection function can be translated to code. Generating code of this function in its domain would result in a `for` loop as in Algorithm 1 that browses the rightmost face of each tile (in this example, tile size is 5 in each dimension).

Algorithm 1: Loop nest describing a facet

```
for $j = 5i_2$ to $5i_2 + 4$ do
    for $k = 5i_3$ to $5i_3 + 4$ do
        facet_i[$j$][$k$] = iteration_result(4, $j$, $k$)
    end
end
```

For hyperplane (i, j), the maximum absolute value of the component along the k axis of every dependence vector is 2. Therefore, the data space to be created will consist of the two uppermost (i, j) planes of every tile.

The mapping here is a modulo projection: instead of getting rid of the component along the k axis, this projection replaces the k axis by $k \mod 2$:

$$P_k(i, j, k) = (i, j, k \mod 2)$$

Its domain is the two uppermost planes:

$$D(P_k) = \{(i, j, k) : 3 \leqslant k \mod 5 \leqslant 4\}$$

General case

Consider, according to Table 3.1, a d-dimensional iteration space D, which is contained in a vector space having an orthonormal basis $(\vec{e_1}, \ldots, \vec{e_d})$. Consider Q uniform dependence vectors $\vec{b_1}, \ldots, \vec{b_Q}$, such that rectangular tiling is legal on all dimensions[1]. Let $s_1, \ldots, s_d$ be the tile sizes.

We notably make the hypothesis that all dependences are shorter than the tile sizes across all dimensions, i.e $|b_j| < s_j$ for all $j \in \{1, \ldots, d\}$. We also assume that all dependence vectors are backwards in all dimensions: $\forall i, j : \vec{b_i} \cdot \vec{e_j} \leqslant 0$, without loss of generality (up to a change of basis).

Let $k \in \{1, \ldots, d\}$, $\vec{e_k}$ the canonical vector of the k-th dimension. The k-th *face* is given by those iterations which k-th coordinate is equal to $t_k - 1$.

In the general case, i.e. for d-dimensional spaces, we determine the thickness of each facet the same way as above: it is the longest dependence vector along the direction normal to that face. Assuming dependence vectors $\vec{b_1}, \ldots, \vec{b_p}$, the thickness of facet normal to $\vec{e_k}$ is given by:

$$w_k = \max_{q \in \{1, \ldots, p\}} \left| \vec{e_k} \cdot \vec{b_q} \right|$$

A tile of iterations, per Table 3.1, is

$$T(\vec{t} = i_1, \ldots, i_d) = \{\vec{x} = (x_1, \ldots, x_d) : \forall q \in \{1, \ldots, d\} : i_q s_q \leqslant x_q < (1 + i_q) s_q\}$$

The k-th facet for tile $\vec{t}$ is the set of iterations given by:

$$F_k(\vec{t}) = \left\{ \vec{x} = (x_1, \ldots, x_d) \in T(\vec{t}) : (1 + i_k) s_k - w_k \leqslant x_k < (1 + i_k) s_k \right\}$$

[1]Rectangular tiling is a special case of tiling, where the tiling hyperplanes are such that their normal vectors are the basis vectors of the space.

To map the facet's iterations to the data spaces, it is then sufficient to take canonical *modulo* projections:

$$P_k(x_1, \ldots, x_d) = (x_1, \ldots, x_{k-1}, x_k \mod w_k, x_{k+1}, \ldots, x_d)$$

We claim that all of the intermediate results needed to execute a tile are contained within facets. To prove this, we need to prove that all the flow-in data for any given tile is contained within facets, and likewise, facets contain all of its flow-out data. By definition, flow-out data for a tile is flow-in data for another tile; therefore, proving the completeness of the flow-in side also proves the flow-out side.

Proposition 1. *For any given tile of coordinates $\vec{t}$, the flow-in iterations of $T(\vec{t})$ are contained inside facets.*

Proof. The iteration-wise flow-in set of a tile is defined as those iterations which result is used by this tile but are executed in another tile. It can be written as:

$$\varphi_i(T(\vec{t})) = \left\{ \vec{y} \in D \setminus T(\vec{t}) : \exists j \in \{1, \ldots, Q\} : \vec{y} - \vec{b_j} \in T(\vec{t}) \right\}$$

Let $\vec{y} \in \varphi_i(T(\vec{t}))$. We will show that $\vec{y}$ belongs to at least one facet of the tile it is contained in.

Let $j \in \{1, \ldots, Q\}$ such that $\vec{y} - \vec{b_j} \in T(\vec{t})$. We know that for all $k \in \{1, \ldots, t\}$, $\left|\vec{b_j} \cdot \vec{e_k}\right| \leqslant w_k$ by definition.

Let $\vec{b_j} = (b_{j,1}, \ldots, b_{j,d})$, and likewise $\vec{y} = (y_1, \ldots, y_d)$. Given that $\vec{y} - \vec{b_j} \in T(\vec{t})$, we know that for every $q \in \{1, \ldots, d\}$:

$$i_q s_q \leqslant y_q - b_{j,q} < (1 + i_q) s_q$$

which means:

$$i_q s_q + b_{j,q} \leqslant y_q < (1 + i_q) s_q + b_{j,q}$$

and because $b_{j,q} \leqslant 0$,

$$i_q s_q - |b_{j,q}| \leqslant y_q < (1 + i_q)s_q - |b_{j,q}|$$

It is impossible that for all $q \in \{1, \ldots, t\}$, $i_q s_q \leqslant y_q$, otherwise $\vec{y} \in T(\vec{t})$. Therefore, let q_0 be such that $i_q s_q - |b_{j,q_0}| \leqslant y_{q_0} < i_{q_0} s_{q_0}$.

Given that dependences are shorter than tile sizes along all dimensions, we know that $\vec{y}$ is in an immediately neighboring tile of $T(\vec{t})$; let $\vec{t'}$ be the coordinates of that tile. We know that the q_0-th coordinate of $\vec{t'}$ is $i_{q_0} - 1$. Therefore:

$$((i_{q_0} - 1) + 1)s_q - |b_{j,q_0}| \leqslant y_{q_0} < ((i_{q_0} - 1) + 1)s_{q_0}$$

which means that $\vec{y} \in F_{q_0}(\vec{t'})$, i.e. $\underline{\vec{y} \text{ is contained within a facet from tile } \vec{t'}}$. $\square$

4.3.5 Flow-in from first-level neighbors: Full-tile contiguity

We have determined that facets contain all the intermediate results necessary to execute tiles. Facets with results are transferred to memory once a tile's execution is complete. To this effect, we must give an allocation to the data contained within each facet.

We want that for each tile, every facet (as in Figure 4.5) is mapped to a contiguous region of memory, to be written with a single burst access. We call this *full-tile contiguity*, and use data tiling to obtain it. The idea is to mirror the iteration space tiles on the data space so that each data tile corresponds to a facet.

This transformation improves both write and read performance: Flow-out facets are almost entirely used by the tile they are immediately adjacent to (as in Figure 4.5).

We give below an example and the general way to apply data tiling in the projected data spaces.

Example

We can continue with the example of Figure 4.3. In this example, using N_i, N_j and N_k as the iteration space size along each axis, we have built data spaces for each facet:

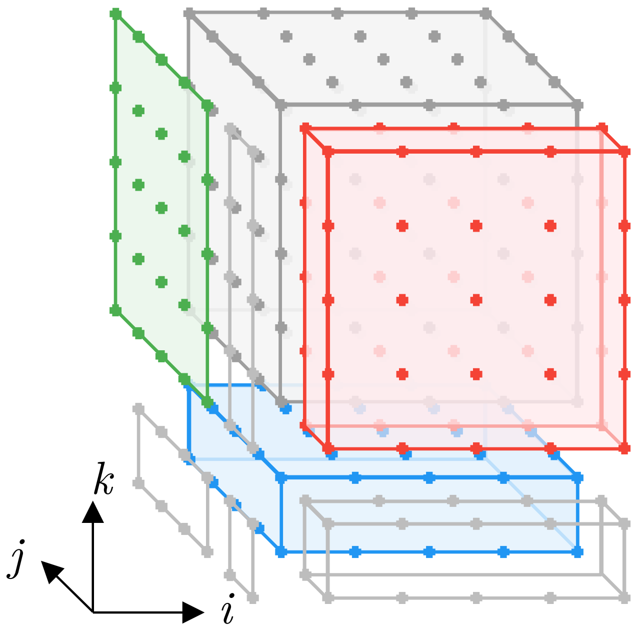


Figure 4.5: First-level neighbor facets that are consumed by the grey tile.

- `facet_i`[N_i/5][N_j][N_k]
- `facet_j`[N_j/5][N_i][N_k][2]
- `facet_k`[N_k/5][N_i][N_j][2]

Hyperplane (j, k) is written to `facet_i` using the projection p_i (plane on the right of Figure 4.4). We seek to further divide `facet_i` to make the footprint of each tile a contiguous block of memory.

The array `facet_i` will therefore be split into tiles of size 5×5, resulting in the following array:

`facet_i`[N_i/5][N_j/5][N_k/5][5][5]

General case

To mirror iteration tiles on the data space, we apply data tiling with the same tile sizes as the iteration space: the projection normal to $\vec{e_k}$ is therefore tiled with dimensions $(t_1, \ldots, t_{k-1}, t_{k+1}, \ldots, t_d)$.

To tile the i-th dimension of an array with size t, we replace the i-th dimension by two dimensions: the quotient and remainder of the Euclidean division of the original dimension by t. This operation gives $2(d-1)$-dimensional data spaces from a $(d-1)$-dimensional projection of the iteration space. All the quotient dimensions are moved first, and the remainder dimensions are moved last. We call the quotient dimensions the ***outer* dimensions**, and the remainder dimensions the ***inner* dimensions**.

The following mapping tiles a 3-dimensional data space A with tile sizes (s_1, s_2, s_3):

$$A[i][j][k] \mapsto A'[i/s_1][j/s_2][k/s_3][i\%t_i][j\%t_j][k\%t_k]$$

This transformation is composed with the projection function of the previous section to give the actual allocation.

Data tiling only guarantees the entire facet is read or written as a single burst access. In the flow-in data from second- or third-level neighbors, as in Figure 4.6, we need to read only a subset of some facets.

To enable cross-tile contiguity in multiple directions, we want adjacent facets from adjacent tiles in the iteration space to also be adjacent in memory. Section 4.3.6 changes the layout of the arrays obtained via data tiling to enable this adjacency.

Choice and cost of memory allocation

It is possible to allocate the same memory cells to facets from distinct tiles, at the condition that the facets do not need to be *live* simultaneously, i.e. that facet data being overwritten will no longer be needed.

Allocations considering these write-after-read dependences, such as lattice-based projection [22], can be used to obtain a legal allocation with a small memory footprint. However, these do not guarantee the possibility of inter-tile contiguity, because they do not necessarily preserve the adjacency of two tiles in the iteration space in the memory allocation of their facets.

In this work, we prioritize contiguity over memory footprint size. To guarantee the existence of inter-tile contiguity, we choose to allocate a *distinct* cell for each facet for each tile, resulting in a *single-assignment* allocation. Although this choice is not optimal, any reduction in memory size performed must keep

The memory cost of this choice is proportional to the size of the iteration space: the total amount of memory needed, considering tile sizes s_i and iteration space sizes N_i on each dimension ($i = 1, \dots, t$), is the volume of each facet (that can be seen in Figure 4.5) multiplied by the number of tiles in the iteration space:

$$S = \sum_{i=1}^{t} [\underbrace{w_i \prod_{\substack{j=1 \\ j \neq i}}^{t} s_i}_{\substack{\text{volume of} \\ \text{facet } i}} \quad \underbrace{\prod_{j=1}^{t} \left\lceil \frac{N_i}{s_i} \right\rceil}_{\substack{\text{number of tiles in} \\ \text{each dimension}}} \quad]$$

4.3.6 Flow-in from second-level neighbors: Inter-tile contiguity

In the general case, part of the flow-in iterations of a tile are located in its second-level neighbors, such as those as shown in Figure 4.6.

Intuitively, second-level neighbors are neighbors of first-level neighbors. We therefore seek to have contiguous accesses crossing data tile boundaries, or *inter-tile contiguity*: we extend burst accesses reading from a data tile from a first-level neighbor to span into a second-level neighbor.

This can be obtained by swapping the dimensions of the data arrays. The next paragraphs explain how.

Example

We take back the example of Figure 4.3, of which the part of the flow-in data in second-level neighbors is shown in Figure 4.6. The producer iterations are part of two facets at the same time. We therefore have to:

1. Choose a direction of contiguity for each facet, and

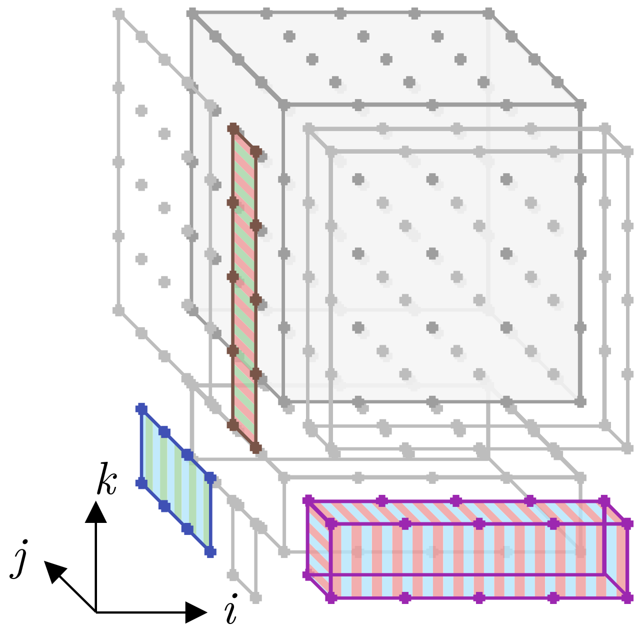


Figure 4.6: Flow-in iterations from second-level neighbors of an iteration tile. CFA places in memory each data tile containing a slab next to a fully consumed data tile to read both in the same burst.

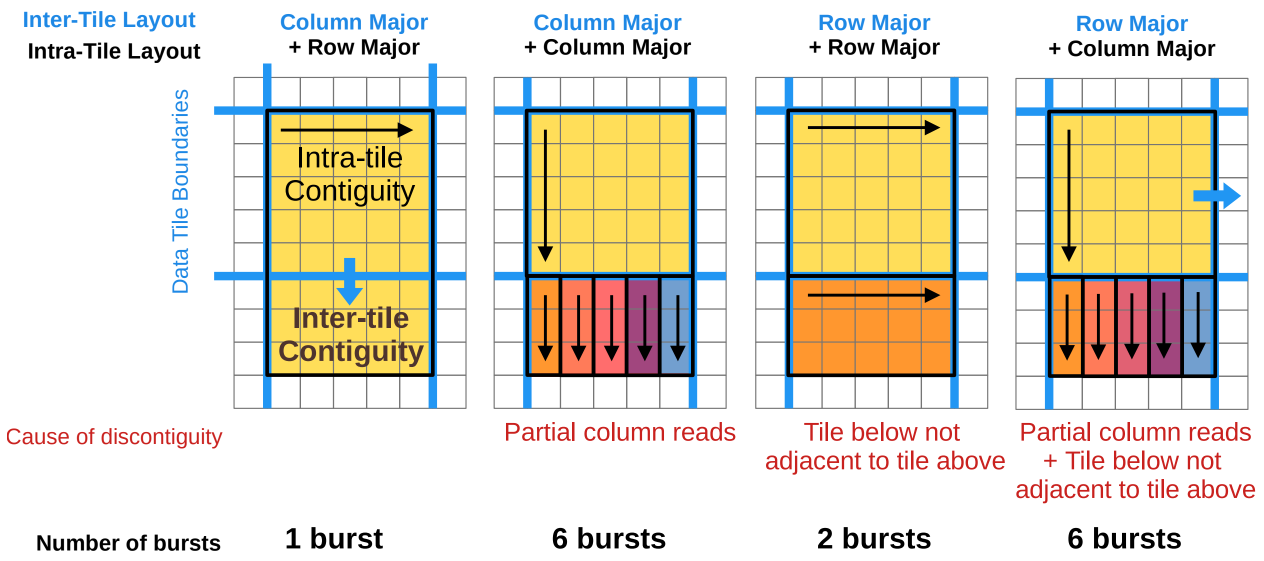


Figure 4.7: The four possible data layouts for a 2-dimensional array in tiled data layout (inter-tile + intra-tile layouts), and the number of bursts needed to read a tile and part of the tile below it. Only one layout allows a single burst spanning both tiles.

2. Select the right facet to read each extension from.

If we call $\Phi_i(i_1, i_2 - 1, i_3 - 1)$ the set coming from tile $(i_1, i_2 - 1, i_3 - 1)$ (purple slab in Figure 4.6), we notice that this slab is a subset of both the (i, j) and (i, k) hyperplanes, so we can read it from both `facet_j` and `facet_k` arrays.

We choose the direction of contiguity for the data space projected from the (i, k) hyperplane to be the k axis, and to read the extension $\Phi_i(i_1, i_2-1, i_3-1)$ from `facet_j`. Figure 4.7 shows the four possible layouts for the `facet_j`: only column-major as a data tile layout and row-major as an intra-tile layout will allow reading $\Phi_i(i_1, i_2 - 1, i_3 - 1)$ as a contiguous extension of `facet_j` from tile $(i_1, i_2 - 1, i_3)$. We swap the dimensions of the `facet_j` array to match it. Column-major inter-tile layout means that dimensions are ordered this way: `i3`, `i1`; and row-major intra-tile layout gives the `i`, `k` order.

Accessing the `facet_j` array is therefore done with `facet_j[i2][i1][i3][k][i][2]`.

The result is that the purple and red slabs of Figure 4.3b can be read in a single, merged burst, from `facet_j`: they are contiguous along the k axis. The same process can be repeated with the other facets.

General case

A tiled data space, as in Figure 4.7, has two dimensions corresponding to each canonical axis (one for tile coordinates, and one for intra-tile coordinates). The direction of inter-tile contiguity for a facet has to be chosen among those axes that are projected.

For a given projection, to make tiles along the i axis contiguous, then the `i1` dimension (outer dimension on the i axis) is moved as the last of the outer dimensions to be enumerated. Cross-tile contiguous reads require the `i` dimension to be the first of the inner dimensions to be enumerated.

Once the direction of inter-tile contiguity is picked, those parts of the flow-in sets that have been made contiguous can be merged to be read in a single burst.

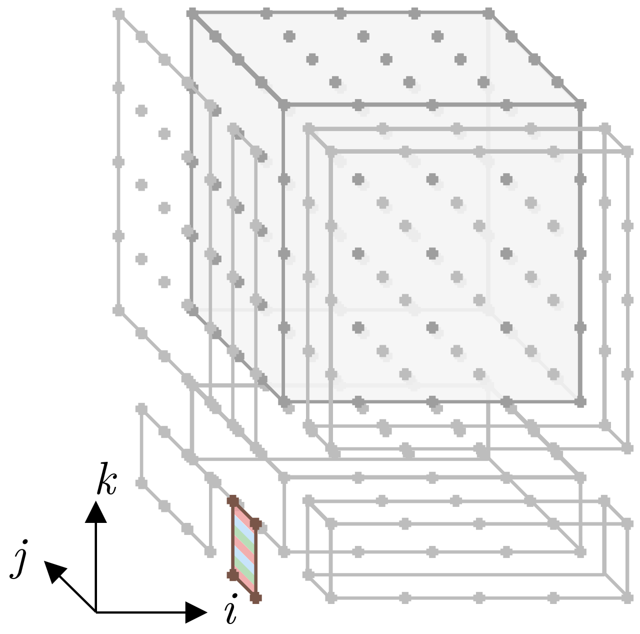


Figure 4.8: Flow-in from third-level neighbor (pattern of Figure 4.3. CFA places these four points contiguously in memory within a data tile (intra-tile contiguity).

4.3.7 Contiguity of flow-in from third-level neighbors

The subsets of flow-in coming from third-level neighbors, as is the set in Figure 4.8, are in general not contiguous in memory to data already accessed from first- or second-level neighbors. Still, this set may be read in a single burst access.

In the specific case of a 3-dimensional iteration space, there is only one flow-in set from a single third-level neighbor. It has a constant number of points, only a function of the dependence pattern. We call S_3 this subset.

For instance, in Figure 4.8, S_3 is the subset of iterations coming from tile $(i_1 - 1, i_2 - 1, i_3 - 1)$ with $i = 4$, $j \in \{3, 4\}$ and $k \in \{3, 4\}$. This subset needs to be contiguous within one of the three projected facets.

It is possible to make S_3's data contiguous in memory by changing the order of the inner dimensions of `facet_k` (projection of (i, j) planes). Since this facet only contains iterations with $k \in \{3, 4\}$, we make S_3 contiguous using the order of dimensions i_3, i_2, i_1, i, j, k. Then, the subset of Figure 4.8 is contiguous within `facet_k`: for any i, the points $(i, 3, 3)$, $(i, 3, 4)$, $(i, 4, 3)$ and $(i, 4, 4)$ are consecutive in memory. We can thus fetch them in a single burst.

The final layout of our arrays is therefore:

- `facet_i[i1][i3][i2][j][k]`
- `facet_j[i2][i1][i3][k][i][j % 2]`
- `facet_k[i3][i2][i1][i][j][k % 2]`

4.3.8 Case of k-th level neighbors

Although full, inter and intra-tile contiguity are always possible for first, second and third-level neighbors in a 3-dimensional space, it may not be in a 4 or higher-dimensional iteration space: the number of k-th level neighbors of a tile is C_k^d, which is higher than d, the maximum number of projections (i.e. of directions of contiguity).

4.4 Implementation

The previous section has introduced how the memory layout we suggest is computed from the polyhedral representation of a program. To use it in an accelerator, we need to generate access code that honors this layout.

This section explains a proof-of-concept source-to-source compiler pass, targeting HLS engines, that applies our layout to inputs and outputs of an FPGA accelerator. This pass performs two main steps: analyzing dependencies and calculating facets, and then, transforming the program to use them. The code resulting code is mapped to an FPGA accelerator using a synthesis tool such as Vitis HLS or Catapult. These steps are explained in the next subsections.

4.4.1 Overview: compiler pass

The flow of our compiler pass is shown in Figure 4.9: it takes the polyhedral representation of a program as input, under the hypotheses stated in the previous section. It determines what the facets are, under the form of sets of points. It then generates loops to scan these

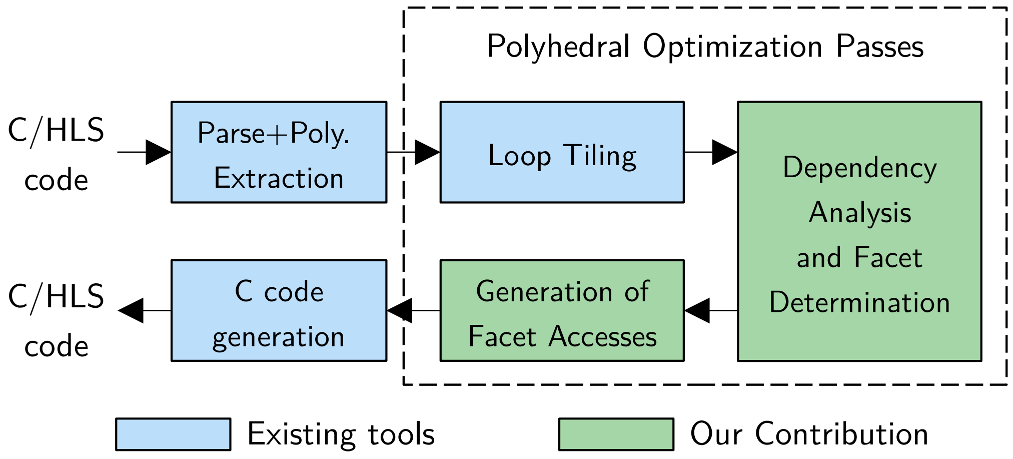

Polyhedral Optimization Passes
C/HLS code
Parse+Poly. Extraction
Loop Tiling
Dependency Analysis and Facet Determination
C/HLS code
C code generation
Generation of Facet Accesses
Existing tools
Our Contribution

Figure 4.9: Polyhedral compiler flow including the CFA pass (shown broken into two).

sets of points contiguously (copy-in / copy-out code), and wraps the tile's code with this copy-in and copy-out code. The copy-in/out code accesses global memory in CFA layout and turns it into the original program's layout for fast on-chip access.

The following subsections explain the transformations that are applied to the code and how copy-in/out code is generated.

4.4.2 Determining the facets and their layout

The technique described in Section 4.3 is applied in order to determine the facets and their layout: given the dependence pattern, the compiler generates the mappings between iterations and data arrays (memory spaces).

4.4.3 Copy-in/out code generation

Once the layout of flow-in / flow-out data is determined by the mappings from the CFA pass, we have to generate code that actually transfers this data using burst accesses. This subsection describes how we proceed.

Selecting flow-in facets

We have proved earlier that the flow-in data of any given tile is contained within facets. However, it is not necessary to read all facets produced by all neighboring tiles to get the flow-in data; only those facets adjacent to the tile to be executed need to be read. We need to refine our reasoning and determine exactly which facets, or subsets of facets, are indeed adjacent in that sense and need to be read.

The following proposition gives us, for each neighboring tile, the facets it produces that contain flow-in data for the current tile.

Proposition 2. *Let $\vec{t} = (i_1, \ldots, i_d)$ and $\vec{t'} = (i'_1, \ldots, i'_d)$ be two neighboring tiles. The flow-in data of $T(\vec{t})$ inside $T(\vec{t'})$ is contained into the following intersection of facets:*

$$\bigcap_{\substack{k=1 \\ i_k \neq i'_k}}^{d} F_k(\vec{t'})$$

Proof. Let $\vec{t} = (i_1, \ldots, i_d)$, and $T(\vec{t})$ be a tile of iterations, and $\vec{t'} = (i'_1, \ldots, i'_d)$ be one of its neighbors. Let $\vec{y} \in \varphi_i(\vec{t}) \cap T(\vec{t'})$ as defined in Section 4.3.3.

Let $\vec{\mu}$ be the difference between tile coordinates:

$$\vec{\mu} = (i_1, \ldots, i_d) - (i'_1, \ldots, i'_d)$$

. $\vec{\mu}$ and $\vec{y}$ are constant throughout all the rest of this proof.

Let $q \in \{1, \ldots, d\}$ such that $\mu_q \neq 0$. We show that $\vec{y} \in F_q(\vec{t'})$. For this, we need to show:

- First, that there is a dependence with a non-null q-th component, i.e. j such that $\vec{b_j} \cdot \vec{e_q} \neq 0$
- Then, that there exists a dependence vector with a non-null q-th component such that translating by its opposite leads outside the tile $\vec{t'}$.

For the second purpose, it is stronger to show that $\vec{y} - \vec{b_j} \in T(\vec{t})$, because it shows at the same time that the q-th facet of $T(\vec{t'})$ contributes to the data flowing towards $T(\vec{t})$.

Per the reasoning used in Proposition 1, considering that the dependences are shorter than tile sizes in all dimensions, we get that $\mu_q = -1$.

Existence of a dependence vector with a non-null q-th component: Assume that $\forall j \in \{1, \dots, p\}, \vec{b_j} \cdot \vec{e_q} = 0$. Then, it is impossible that $\mu_q \neq 0$. Because $\mu_q = -1$, we have

$$(i_q - 1)s_q \leqslant y_q < i_q s_q$$

and therefore if for all j, $\vec{b_j} \cdot \vec{e_q} = 0$, and $y_q + b_j = y_q$ (i.e. there is no way to move along the q-th axis). The q-th coordinate of $\vec{y}$ makes it be outside $T(\vec{t})$. Therefore, it is impossible that $\vec{y} + \vec{b_j} \in T(\vec{t})$ for any j.

Translation to $T(\vec{t})$: Assume that no dependence vector with a non-null q-th component translates $\vec{y}$ into $T(\vec{t})$, i.e. for all j such that $\vec{b_j} \cdot \vec{e_q} \neq 0$, $\vec{y} - \vec{b_j} \notin T(\vec{t})$.

We show this situation is absurd: since $\mu_q = -1$, the q-th component of $\vec{y}$ causes it to be outside $T(\vec{t})$.

- If there exists no dependence vector with a null q-th component, then it is impossible that $\vec{y} \in \varphi_i(\vec{t})$ by definition of the flow-in (there must be at least one dependence vector leading to $T(\vec{t})$), therefore there is a vector with a non-null q-th component which opposite translates $\vec{y}$ into $T(\vec{t})$.

- If there exists a dependence vector with a null q-th component, then let $k \in \{1, \dots, Q\}$ such that $\vec{b_k} \cdot \vec{e_q} = 0$ and $\vec{y} - \vec{b_k} \in T(\vec{t})$. Because $\mu_q = -1$, we have

$$y_q < i_q s_q$$

. The dependence vector $\vec{b_k}$ with a null q-th component will not cause

$$y_q - b_{k,q} \geqslant i_q s_q$$

and therefore $\vec{y} - \vec{b_k} \notin T(\vec{t})$, which is absurd.

In all cases, we get that: if $\mu_q \neq 0$, then there exists a dependence vector $\vec{b_k}$ with a non-null q-th component such that $\vec{y} - \vec{b_k} \in T(\vec{t})$, i.e. $\vec{y} \in F_q(\vec{t'})$.

Considering all the non-null μ_qs, we get that

$$\vec{y} \in \bigcap_{\substack{q \in \{1,\ldots,d\} \\ \mu_q \neq 0}} F_q(\vec{t'})$$

Given that $i_k \neq i'_k$ is equivalent to $\mu_k \neq 0$, the flow-in of tile $T(\vec{t})$ contained within $T(\vec{t'})$ is contained within the intersection of all the k-th facets such that $i_k \neq i'_k$:

$$\bigcap_{\substack{k=1 \\ i_k \neq i'_k}}^{d} F_k(\vec{t'})$$

$\square$

Making a contiguous flow-in/flow-out access pattern

We need to generate an access pattern that will take out the flow-in data from facets, and conversely, write the flow-out data into facets. This access pattern is supposed to be as contiguous as possible, and we expect the longest possible burst accesses.

Flow-out facets are entirely written with one transaction per facet - all the flow-out data is contained.

The intersection of the actual flow-in set with each facet may not be exactly a rectangle, and is perhaps not contiguous inside that facet. For this reason, a rectangular over-approximation of the set of accessed data is taken, like in Figure 4.10. That superset may span across facets from multiple iteration tiles - in this case, cross-tile contiguity ensures that a single transaction can bring all the data on-chip.

Given that a bounding box of a subset is contained in the bounding box of the containing set, doing so incurs less overhead than making a rectangular bounding box of the whole flow-in data.

Correctness: For flow-out data, due to the choice of using a tile-wise single-assignment allocation (no two different tiles write in the same memory cells), this over-approximation

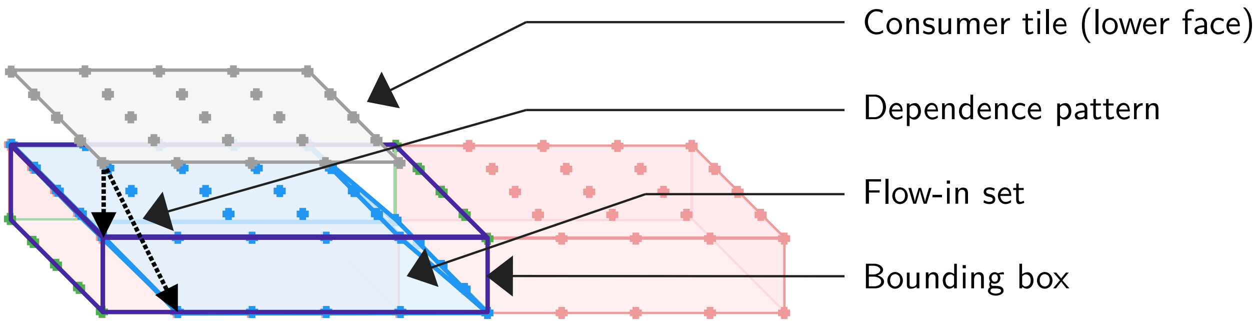


Figure 4.10: Taking a rectangular over-approximation of the actual flow-in set incurs redundant reads. Contiguous data (facets) are in red: an over-approximation may be necessary to preserve access contiguity.

poses no correctness issues. For flow-in accesses, we must ensure this over-approximation doesn't break correctness (conflicts caused by two iterations sharing the same on-chip cell, while one is not part of the flow-in data). Filtering out the unneeded data coming from the bounding box is therefore necessary by adding a guard to the copy-in code.

Generating Burst-Capable Code

The code to be generated to fetch and write flow-in/flow-out sets has to trigger burst accesses on its memory controller. We generate synthesizable high-level synthesis (HLS) code, where bursts are inferred from `for` loops. The following conditions are sufficient for a burst to be inferred:

- The number of addresses accessed is explicit (e.g., the trip count of the copy loop nest is constant),
- If the target array size is known, the memory addresses are contained within its bounds,
- The addresses accessed are consecutive (e.g., with a pointer increment),
- The copy loop is pipelined with an initiation interval equal to 1.

Using the rectangular over-approximation of facet accesses, we can generate rectangular loop nests, that are coalesced into a single loop to force the inference of a single burst instead of a series of shorter bursts, as in Figure 4.11.

```
copy:
float* offChipAddr = &facet_0[l][n][m][0] + 0;
for(int I = 0; I <= 9215; I = I + 1) {
  #pragma HLS PIPELINE II=1
  int c6 = I / 96 % 96;
  int c7 = I % 96;
  *offChipAddr = A_local[(96*n + c7 + 0) % 128][(96*m + c6) % 128][(96*l
    + 95) % 128];
  offChipAddr = offChipAddr + 1;
}
```

Figure 4.11: Merger of two loops from which a burst access is inferred. On-chip addresses are random, while off-chip addresses are consecutive.

The copy code features two address generators, for off-chip and on-chip data. Off-chip address calculation is straightforward: when accesses are contiguous, one simply needs to provide the HLS tool with a pointer that starts at the beginning of the memory region to be accessed, and increment it. On-chip address calculation is performed at each cycle.

The resulting code for each facet is as in Figure 4.11, and is sufficient to get burst accesses using a commercial HLS tool like Vitis HLS.

4.4.4 Generating HLS code

The final step in CFA code generation is to generate a three-step coarse-grain pipeline:

- The first step reads the flow-in data from global memory in CFA allocation, and turns it into local allocation,
- The second step is tile execution,
- The last step is writeback from the accelerator to global memory.

This is implemented under the form of a function, as in Figure 4.12. Note the DATAFLOW pragma, which is used by the HLS engine to generate a coarse-grain pipeline where the three functions `read`, `execute` and `write` are executed in parallel, each function for a different tile.

```
void toplevel(int i, int j, int k, float* facetIJ,
  float* facetIK, float* facetJK) {
#pragma HLS INTERFACE m_axi port=facetIJ
#pragma HLS INTERFACE m_axi port=facetIK
#pragma HLS INTERFACE m_axi port=facetJK
#pragma HLS DATAFLOW
    float buf1[TS*TS];
    float buf2[TS*TS];
    read(i, j, k, facetIJ, facetIK, facetJK, buf1);
    execute(i, j, k, buf1, buf2);
    write(i, j, k, buf2, facetIJ, facetIK, facetJK);
}
```

Figure 4.12: Top-level function, assuming tile size is `TS`, local memories are `TS`^{2} big and the iteration space is `TS`^{3}. Such sizes are typical of iterative stencils.

4.5 Evaluation

We evaluate the effects of adding CFA allocation to an FPGA accelerator with respect to two metrics: bandwidth (relative to state-of-the-art allocations, and with respect to the bus peak bandwidth), and FPGA resource overhead introduced by the address generation code.

4.5.1 Experimental protocol

Experiments were carried out on a variety of uniform-dependence benchmarks listed in Table 4.1[2]. Iterative stencils update an array in place, and differ by their dependence pattern and the dimensions of the iteration space.

The platform used is a *Xilinx Zynq ZC706* board, including a *xc7z045ffg900-2* FPGA. The test accelerators only comprise read and write parts, the structure being that of Figure 4.13. Every baseline has been tested by connecting it to a single AXI high-performance port mapped to DRAM (port HP0); the frequency of every design is 100.00 MHz, the AXI bus is 64-bit wide, and the data type transferred over the bus is 64-bit double-precision IEEE floating point numbers.

[2]Results for `jacobi2d9p-gol` are omitted from this manuscript; full results are available at `https://arxiv.org/abs/2202.05933`

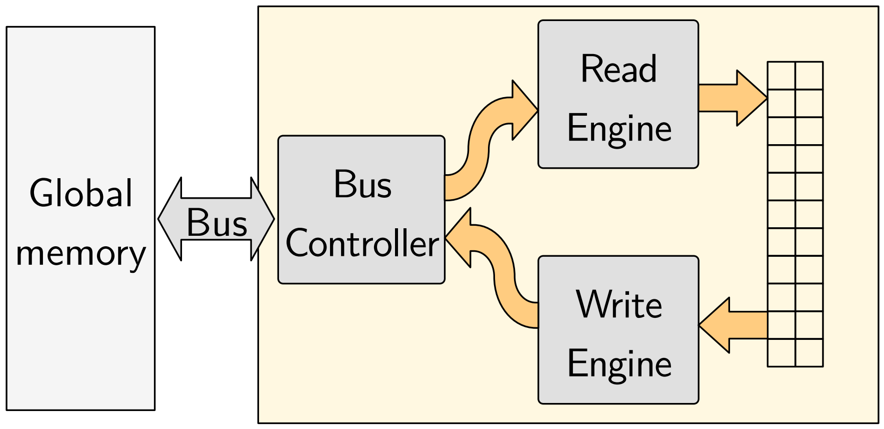


Figure 4.13: Memory-bound accelerator used for benchmarking: only the read and write engines are implemented.

Table 4.1: Benchmarks used for testing CFA. Equivalent applications have the same computational dependence pattern as the benchmark and would show similar performance.

Dependence pattern	Nb of deps	Tile Sizes	Equivalent Application
`jacobi2d5p`	5	$16^3 \rightarrow 128^3$	Laplace equation
`jacobi2d9p`	9	$16^3 \rightarrow 128^3$	3×3 convolution
`jacobi2d9p-gol`	9	$16^3 \rightarrow 128^3$	2nd-order finite difference
`gaussian`	25	$4 \times 16^2 \rightarrow 4 \times 128^2$	5×5 Gaussian Blur
`smith-waterman -3seq`	7	$16^3 \rightarrow 128^3$	Alignment of 3 sequences

Baselines

We considered the following baselines for comparison with CFA:

- **Original Layout** (as done by Bayliss et al. [5]): a best-effort burst access pattern is determined under the original allocation. This access pattern does not issue any redundant reads, possibly at the expense of contiguity.
- **Bounding Box** (as done by Pouchet et al. [60]): a rectangular bounding box around the flow-in and flow-out data is taken so as to exhibit burst transfers; part of the bounding box is unused and redundantly transferred.
- **Data Tiling** (as done by Ozturk et al. [58]): data tiling is applied to the original arrays, and any tile that is accessed is entirely transferred. The reported value corresponds to the best performing tile size that is less or equal to the iteration tile size.

The original layout baseline introduces no transfer redundancy but has the smallest amount of and the shortest burst transfers. The two other baselines are trade-offs between burst usefulness and bandwidth use: using a bounding box or data tiling result in using only long burst accesses at the price of transfer redundancy. Each benchmark is tested against a variety of tile sizes, with 1:1, 1.5:1 and 2:1 ratios.

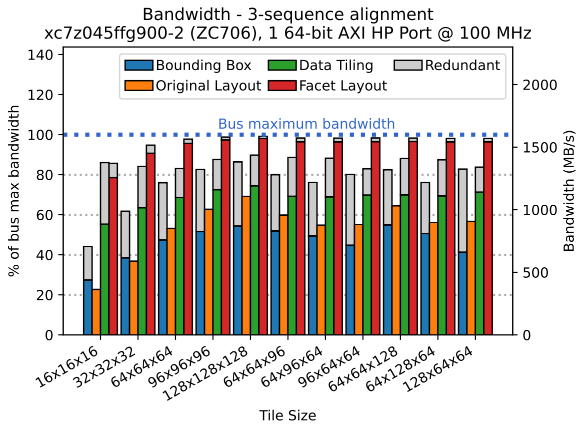
4.5.2 Results and discussion

Raw bandwidth

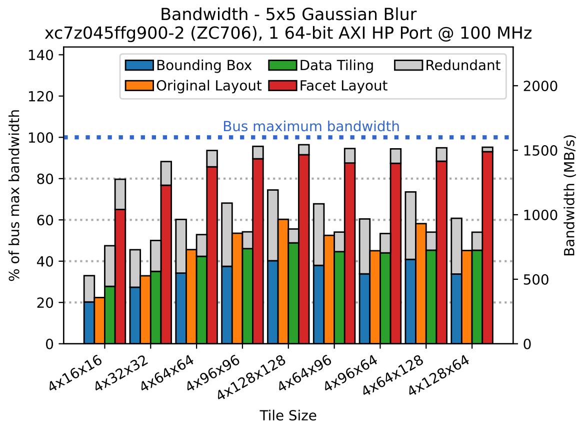
The raw bandwidth shown in Figure 4.14 indicates that the CFA layout and access pattern can reach close to 100% of the bus' maximum bandwidth, whereas other baselines exhibit high redundancy overhead (especially with the bounding box).

The high efficiency of our approach is mainly explained by three points:

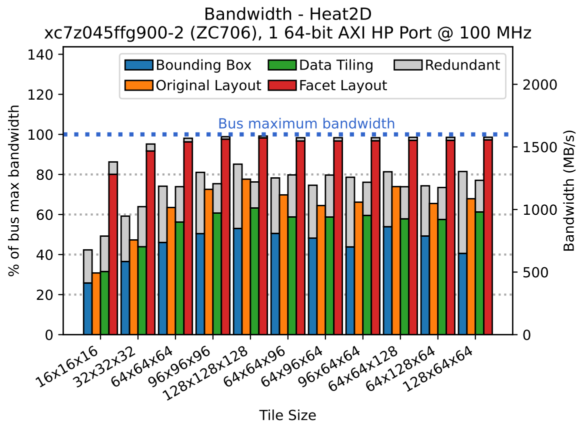
- The small number of burst transfers per tile (4 in the case of 3-dimensional tiles). The data tiling approach uses a single burst access per tile, and the original layout and bounding box approaches issue multiple burst requests.



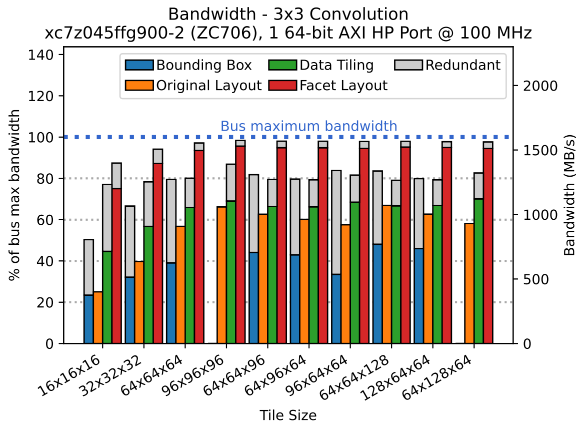
(a) `sw3d`



(b) `gaussian`



(c) `jacobi2d5p`



(d) `jacobi2d9p`

Figure 4.14: Bandwidth for all baselines, per benchmark. CFA is efficient even for tiles where one dimension is much smaller than the others. Bounding box is the technique used by Pouchet et al. [60], Data Tiling is used by Ozturk et al. [58], and the original layout is the best-effort baseline as in Bayliss et al. [5].

- The length of these burst transfers: some accesses in CFA are entire facets,
- The ability of Vitis HLS to use burst access overlapping, which hides latency for long bursts even when they are decomposed into smaller burst accesses.

Effective bandwidth

The effective bandwidth assesses the usefulness of the transferred data: it only counts the data transferred that is actually useful for the application. Data transferred then ignored is consuming bus time, thus lowering the effective bandwidth. Figure 4.14 shows the effective bandwidth with colors, the difference with raw bandwidth being in gray. Two observations can be made:

- For the considered tile sizes, CFA is able to bring the effective bandwidth close to 100% of the bus bandwidth, which other allocations will not achieve.
- CFA is efficient even with small tile sizes. The `gaussian` benchmark, tiled with a small size in time (4) and larger spatial sizes (up to 128×128), shows that CFA exceeds 80% of the bus bandwidth for tile sizes above $4 \times 64 \times 64$.

The high usefulness of CFA (low difference between effective and raw bandwidth) is due to the choice of projections in CFA, which yields minimal redundancy.

Area cost

We analyze two distinct cost metrics specific to FPGA designs: the computational resources (Slice and DSP), and the storage resources (Block RAM).

Computational resources **The cost of CFA itself in terms of hardware is the address generators. These are small, as about 95% of the logic area on our test platform remains available for compute engines.**

Regardless of the off-chip and on-chip allocations, the read and write engines take up a small fraction of the available logic resources. Figure 4.15 aggregates the area occupied by all baselines other than CFA for all tile sizes we have tried, and positions CFA. It can be observed that with tile sizes ranging from 16^3 to 128^3, except `gaussian` which tile sizes range from 4×16^2 to 4×128^2, designs occupy between 2 and 5% of the total slice area, and 0 to 3% of the total available DSP resources on a XC7Z045 FPGA chip. Canonical Facet Allocation does not show a significantly different slice occupancy than other baselines. CFA requires DSP blocks to compute off-chip base addresses, but we observe that none of our benchmarks exceeded 40 out of 900 (4%) of the DSP resources.

We observed the worst usage of DSP resources on non-power of two tile sizes; computing the base address of a data tile takes a DSP block for a true integer multiplication. Additionally, the experiments were carried out without any compute logic on the accelerators; given the small size of the memory access modules, the synthesis tool did not have to significantly optimize the design for area.

Block RAM usage Using CFA does not change the on-chip allocation, therefore using CFA does not significantly increase the BRAM cost of a design.

In an FPGA design, Block RAM resources used for on-chip data storage are shared between multiple actors. Even when the compute actor is not implemented, the memories needed to hold all the data on chip in and out of the memory actor must be present. Therefore, all designs except `gaussian` occupy up to 95% of the available on-chip Block RAM, as Figure 4.16 shows. BRAM was, indeed, the factor limiting tile size - the larger the tile, the more data needs to fit into on-chip memories. The `gaussian` exception is due to one of its tile sizes being constant and small.

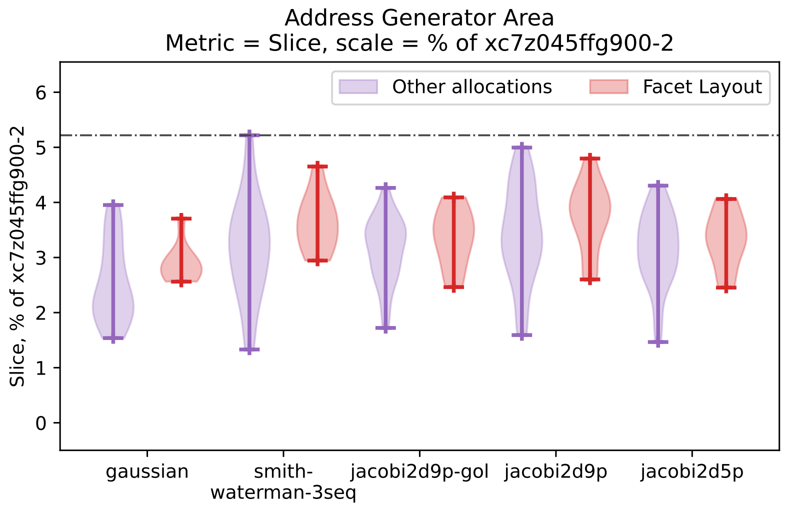
We can observe in Figure 4.16 that the distributions of on-chip memories using CFA and the original allocation are the same, with an exception (`smith-waterman-3seq`). This is due to the fact that CFA does not change the on-chip allocation, which is defining the amount of on-chip memory needed. The BRAM overhead for bounding box and data tiling baselines is mainly due to two facts: for the bounding box baseline, writing a superset of the tile footprint implies that the values written while not modified have been read and held on chip.

4.6 Conclusion

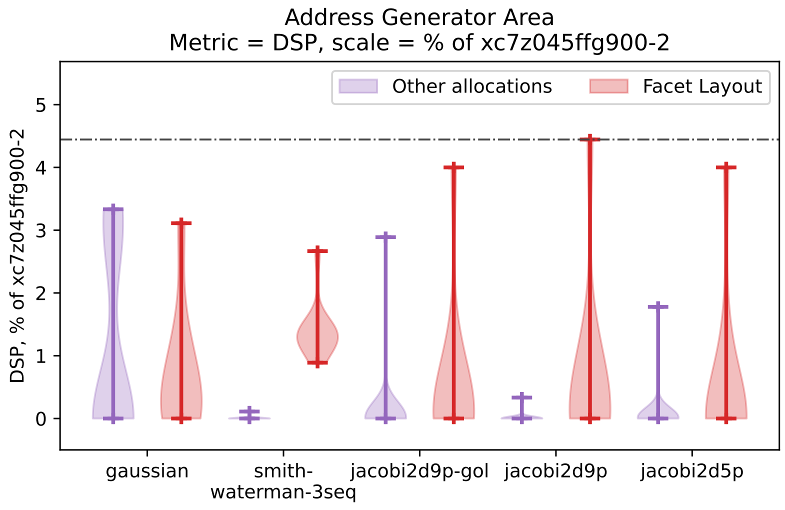
Our work provides an answer to the under-utilization of memory bandwidth observed in many instances where an FPGA or ASIC accelerator is developed. The insufficient effective memory bandwidth results in stalls, preventing the full exploitation of the on-chip parallelism. Memory allocation can be the cause of a significant under-utilization of the available bandwidth, due to the high latency of scalar accesses.

To overcome such under-utilization of bandwidth, we introduced the Canonical Facet Allocation, a burst-friendly data layout in memory as well as a compiler pass that transforms code to use it. It enables the use of the full bus bandwidth, for a low logic overhead.

To further increase the benefits of CFA, the machine model we have considered may be extended to multi-port memory accesses, such as high-bandwidth memory, and distributed memories. In such architectures, there are multiple data ports; to benefit from all their bandwidth, one has to find an adequate repartition of data over each memory port to balance accesses.



(a) Occupancy of logic slices



(b) Occupancy of DSP blocks

Figure 4.15: Area occupied by CFA and all other baselines (aggregated), as a percentage of the available area of xc7z045ffg900-2. The vertical lines span from minimum to maximum.

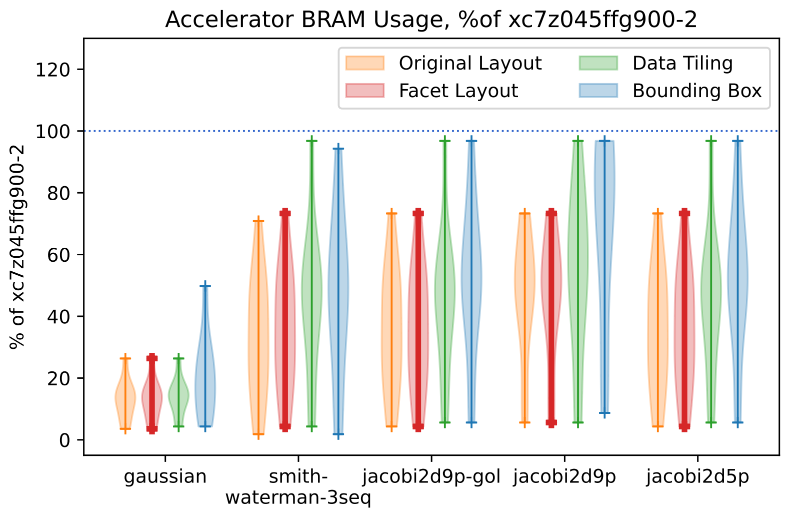


Figure 4.16: BRAMs occupied by all baselines (in color), for each benchmark, as a percentage of xc7z045ffg900-2 BRAMs.

Chapter 5

Maximal Atomic irRedundant Sets: a Usage-based Dataflow Partitioning Algorithm

5.1 Introduction

Chapter 4 has introduced a memory allocation that can be applied to rectangular tiles of a polyhedral program. Thanks to the simplicity of the rectangular shapes, it is possible to obtain a high degree of contiguity, with a limited amount of redundant reads and writes.

With more complex tile shapes that are not eligible for the CFA technique, such as diamond tiles [9], it is desirable to keep redundant accesses as low as possible while keeping a good utilization of the bandwidth. In this chapter, we seek to have the highest utilization of bandwidth under the constraint to have no redundancy.

The work in chapter relies on prior polyhedral analysis and locality optimizations seen in Chapter 3 to have been done. Notably, we use the polyhedral reduced depdendence graph (PRDG), and loop tiling background seen in Section 3.4.3. We assume tiling has been applied to all the programs considered, and focus on inter-tile communications.

In this chapter, we derive sets of data that are communicated between tiles of a polyhedral program, with a strict condition to not allow *redundancy*, both in terms of write (no data is written more than once into memory) and read (no unused data is read from memory).

This chapter brings in the following contributions:

- A method to partition of the flow-in and flow-out of each tile into *Maximal Atomic irRedundant Sets* (MARS),
- A calculator implementing this method, and an evaluation of its results on a selection of polyhedral benchmark programs.

It is organized as follows: Section 5.2 gives a view of other work related to data flow decomposition; Section 5.3 describes the MARS sets and gives their construction procedure; Section 5.4 provides examples of constructed sets and analyses them; Section 5.5 gives possible applications of our work.

5.2 Related Work

The flow-in and flow-out sets have been extensively studied along with a good amount of breakup scenarios by Datharthri et al. (2013) [23] and Bondhugula (2013) [7]. We are focusing on one special case along the lines of the work of Datharthri et al., with a constraint that no point may belong to two communication sets at the same time.

A decomposition of the communicated sets of data may be used for inter-node message passing in MPI to reduce the amount of traffic. Zhao et al. [90] perform a decomposition of the data space of stencils into coarse blocks such that fetch and write operations of each block are contiguous, and blocks are laid out according to the consuming neighbors so that a series of blocks is retrieved in one contiguous message. A supporting graph data structure provides the addresses for each of the blocks. This work seeks an optimal memory layout in terms of number of communications, and does so without the flow-in and flow-out sets or a polyhedral representation. Our work generalizes the idea using the polyhedral framework.

5.3 MARS: Maximal Atomic irRedundant Sets

This section presents the Maximal Atomic irRedundant Sets (MARS). It is laid out as follows: first, we give a definition of MARS and properties that they match. Then, we introduce an algorithm to construct these sets along with an example.

5.3.1 Notations and hypotheses

To compute the MARS, we need a program with a polyhedral representation, to which the tiling transformation is legal along given hyperplanes. We restrict ourselves to the case where the dependences are uniform, and therefore we can consider individual *dependence vectors*. The uniformity of the dependence pattern guarantees that, assuming an infinite iteration space, all tiles of the same shape feature the same MARS. Tile sizes are assumed to be constant, but they can be made runtime parameters using the idea from [64].

We also assume that each statement writes to a single memory location. Therefore, we can interchangeably use an iteration and the value it produces (data).

Additionally, we will use the following notations, per Table 3.1:

- D designates the d-dimensional iteration space.
- There are t tiling hyperplanes, H_i for $i \in \{1, \ldots, t\}$.
- There are Q dependence vectors, $\vec{b_j}$ for $j \in \{1, \ldots, Q\}$. The Polyhedral Reduced Dependence Graph (PRDG) is the set of all dependence vectors, noted $\mathbf{B}$.

The following additional notations are added for this chapter:

- The *non-trivial parts* of a set E are all the non-empty subsets of E. It is noted $\mathcal{P}_n(E)$. For instance:

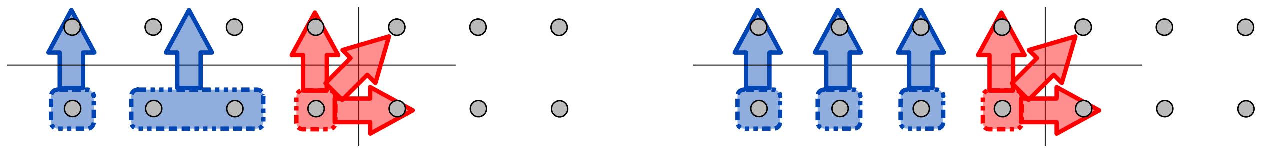
$$\mathcal{P}_n(\{1,2,3\}) = \{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$$

- The modulo operator is noted $a \bmod b$ and congruences are noted $a \equiv r[b]$.

5.3.2 Definition

Maximal Atomic irRedundant Sets (MARS) are defined as the maximal sets of iterations that satisfy the following properties:

- **Single-producer** (SP): Each MARS is produced by a single tile.
- **Atomicity** (A): If any data from a MARS is consumed by a tile, then all points within that MARS are also consumed by the same tile.



(a) a **(b)** b

Figure 5.1: Sets of iterations for Smith-Waterman matching single-producer, atomicity properties, but not maximality. Maximality forces the search for a non-trivial solution

- **Maximality** (M): If two iterations within the same tile have exactly the same consumer tiles, the data they produce belong to the same MARS.

Figure 5.1 shows the need for the maximality property. A trivial decomposition of the entire flow-out set into singletons would satisfy the (SP) and (A) properties, but is not what is sought. The maximality property forces the search for a non-trivial solution.

5.3.3 Computation

The all-consumed property stated above is equivalent to saying that all data inside a MARS is consumed by exactly the same tiles. Therefore, if two distinct tiles consume a MARS M, then given the all-consumed property, both consumer tiles use all of the points of M.

We propose a construction by breaking up the *flow-out set* of each tile. There is an equivalent construction with the *flow-in set* of each tile, and both constructions lead to the maximal sets that respect the three (SP), (A) and (M) properties.

Flow-out set

We start by refining the definition of the flow-out set seen in Figure 4.3, and provide a different construction than [7]. The flow-out set of a tile is defined as all the iterations which have at least one consumer iteration outside the tile. An example is given in Figure 5.2 for a Smith-Waterman kernel with skewed tiles. Notably, we see that despite the dependences being all unit or null along each axis, the "thickness" of the flow-out may be greater than one (in this case, with the diagonal dependence).

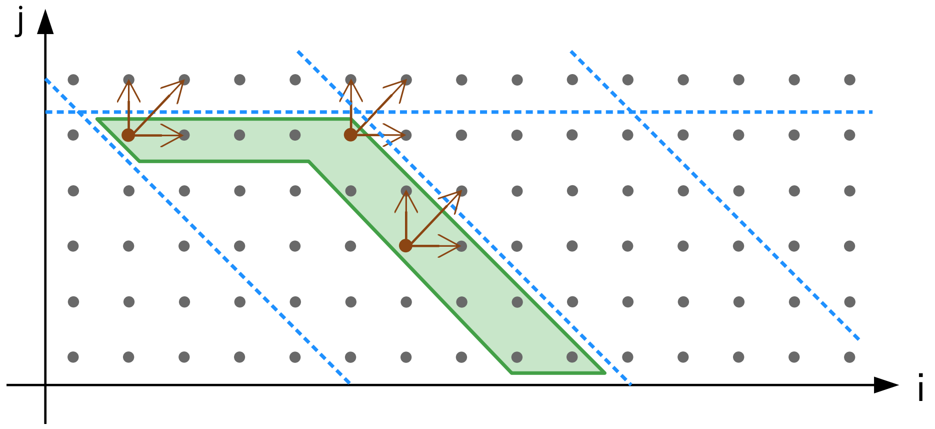


Figure 5.2: Flow-out set for a skewed tile of a Smith-Waterman kernel. The arrows correspond to the dependence pattern (PRDG).

The tile-wise flow-out can be expressed as follows: given a tile $T(\vec{t})$,

$$\varphi_O(T(\vec{t})) = \left\{ \vec{x} \in T(\vec{t}) : \vec{x} + \vec{b}_1 \in D \backslash T(\vec{t}) \vee \cdots \vee \vec{x} + \vec{b}_D \in D \backslash T(\vec{t}) \right\}$$

However, this formulation is missing information on the tile the dependence vectors lead to. We therefore introduce a finer formulation with the individual contribution of each dependence that traverses a tiling hyperplane, i.e. that crosses tile boundaries. This only requires knowledge of the PRDG (dependence vectors) alongside the tiling hyperplanes and the domain, and can be done as in Algorithm 2.

Following Algorithm 2, the flow-out of a tile is then the union of the contributions of all dependences to it:

$$\varphi_O(\vec{t}) = \bigcup_{H \in \mathcal{H}} \bigcup_{\vec{b} \in \mathbf{B}} F_{H,\vec{b}}$$

MARS partitioning of the flow-out set

We explain how our partitioning scheme is done in this subsection.

Principle The flow-out set can be partitioned into MARS in such a way that, for any given tile, all iterations it produces in a MARS have the exact same consumer tiles. The partitioning idea is illustrated in Figure 5.3.

Algorithm 2: Computing the the flow-out set using contributions from each dependence

```
Input: $D$ = iteration space,
       $\mathbf{B} = \left\{\vec{b}_i : i = 1, \ldots, Q\right\}$ = PRDG,
       $\mathcal{H} = \{H_i : i = 1, \ldots, t\}$ = tiling hyperplanes
       $\{s_i : i = 1, \ldots, t\}$ = tile sizes
Result: $\varphi_O(\vec{t})$ = Flow-out set parametrized by tile $\vec{t}$
// $T(\vec{t})$ is a tile of coordinates $\vec{t}$
let $T(\vec{t}) = \{\vec{x} \in D : \forall i \in [\![1; t]\!] : t_i s_i \leqslant \vec{x} \cdot \vec{n}_i < (1 + t_i)s_i\}$;
for $H \in \mathcal{H}$ do
    $\vec{n} = (n_i)_{i=1,\ldots,t}$ normal vector to $H$;
    $s =$ tile size for hyperplane $H$;
    for $\vec{b} \in \mathbf{B}$ do
        // Flow-out iterations for dependence $\vec{b}$ crossing hyperplane $H$
        $m = \vec{n} \cdot \vec{b}$;
        $F_{H,\vec{b}}(\vec{t}) =$
          $T(\vec{t}) \cap \left\{\vec{x} = (x_i)_{i=1,\ldots,d} \in D : \left(-m \leqslant \sum_{i=1}^{d}(n_i x_i) \bmod s < 0\right) \wedge \vec{x} + \vec{b} \in D\right\}$;
    end
end
$\varphi_O(\vec{t}) = \bigcup_{H \in \mathcal{H}} \bigcup_{\vec{b} \in \mathbf{B}} F_{H,\vec{b}}$;
return $\varphi_O(\vec{t})$
```

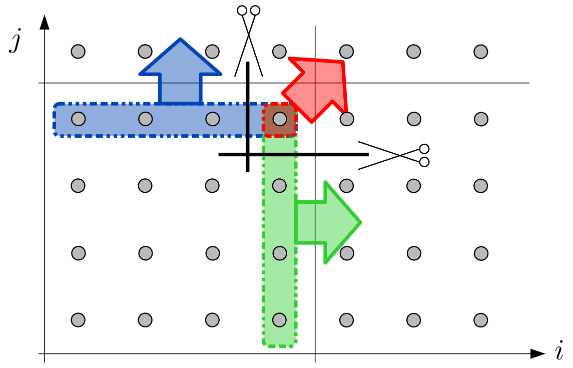


Figure 5.3: Flow-out set of a tile intersected with each consumers tile's flow-in set. The breakup we propose, illustrated by scissors, splits iterations that have different consumer tiles.

The partitioning is done by computing those subsets of the flow-out for which, given select tiling hyperplanes, any dependence crosses all these tiling hyperplanes, and no dependence crosses any other tiling hyperplane.

We browse all possible consumers by applying the above on all combinations of tiling hyperplanes. As there are T tiling hyperplanes, there are $2^T - 1$ possible consumer tiles, and therefore at most $2^{2^T-1} - 1$ MARS.

Example We can construct the MARS for a Smith-Waterman kernel, which has the following characteristics:

- Domain: $D = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} : (i,j) \in [0,100]^2 \right\}$
- PRDG: $\mathbf{B} = \left\{ \vec{b}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \vec{b}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \vec{b}_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$ $(Q = 3)$
- Tiling hyperplane families:

  $\mathcal{H} = \{\{H_1 : i \equiv 0[4]\}, \{H_2 : j \equiv 0[4]\}\}$ $(t = 2)$
- Normal vectors: $\left\{ \vec{n}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \vec{n}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$
- Tile sizes: $\{4, 4\}$

We can first notice that dependence $\vec{b}_1$ does not cross hyperplane H_2, and likewise dependence $\vec{b}_2$ does not cross hyperplane H_1.

As a starting point, we consider those dependences that cross any hyperplane of the H_1 family and none of H_2.

To have a dependence cross hyperplane H_1, the source iteration $\vec{x} = \begin{pmatrix} i \\ j \end{pmatrix}$ must be such that, if $\vec{b} = \begin{pmatrix} b_i \\ b_j \end{pmatrix}$, then $(i \bmod 4) + b_i \geqslant 4$ or $(i \bmod 4) + b_i < 0$. Because all $\vec{b}$s only have positive coordinates, we may just consider the case $(i \bmod 4) + b_i \geqslant 4$.

From dependence $\vec{b}_1$, we get $i \bmod 4 \geqslant 3$; from dependence $\vec{b}_3$, we also get $i \bmod 4 \geqslant 3$. Therefore, the set of points such that *any* dependence crosses H_1 is:

$$\left\{ \vec{x} = \begin{pmatrix} i \\ j \end{pmatrix} \in D : i \bmod 4 \geqslant 3 \right\}$$

or equivalently

$$\left\{ \vec{x} = \begin{pmatrix} i \\ j \end{pmatrix} \in D : i \equiv 3[4] \right\}$$

We compute the subset of these points for which H_2 is crossed. The condition to cross H_2 is that, if $\begin{pmatrix} i \\ j \end{pmatrix} = (\vec{x} + \vec{b})$, then $j \bmod 4 + b_j \geqslant 4$, which means $j \bmod 4 \geqslant 3$ with both dependences $\vec{b}_1$ and $\vec{b}_2$. We therefore get that the points from which H_1 is crossed and not H_2 is:

$$\left\{ \vec{x} = \begin{pmatrix} i \\ j \end{pmatrix} \in D : i \equiv 3[4] \wedge \neg(j \equiv 3[4]) \right\}$$

We can do the same procedure to cross only H_2, and both of H_1 and H_2, which yield the following sets:

$$\left\{ \vec{x} = \begin{pmatrix} i \\ j \end{pmatrix} \in D : \neg(i \equiv 3[4]) \wedge (j \equiv 3[4]) \right\}$$

and

$$\left\{ \vec{x} = \begin{pmatrix} i \\ j \end{pmatrix} \in D : i \equiv 3[4] \wedge j \equiv 3[4] \right\}$$

Those sets are the MARS we were looking for, and correspond to those in Figure 5.4.

Algorithm Algorithm 3 gives the computation procedure to construct all MARS for all tiles.

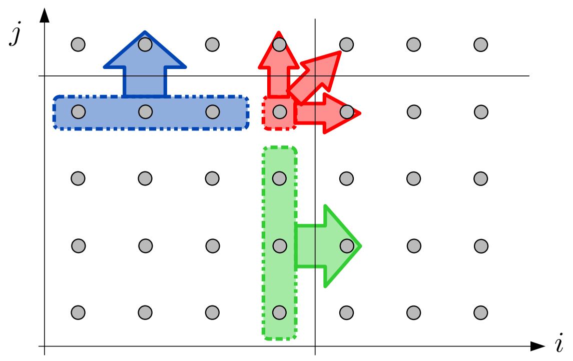


Figure 5.4: MARS and their consumers for Smith-Waterman using square tiling.

In this algorithm, crossing a hyperplane is a shortcut for the property used in Algorithm 2. Assume $\vec{n} = (n_1, \ldots, n_d)$ is the normal vector to a hyperplane H, s is the tile size along that hyperplane, $\vec{b}$ is a dependence vector, and $\vec{x} = (x_1, \ldots, x_d) \in D$. Let $m = \vec{n} \cdot \vec{b}$ assuming $m > 0$. Then:

$$\vec{x} + \vec{b} \text{ crosses } H \Leftrightarrow -m \leqslant \sum_{i=1}^{d} (n_i x_i) \bmod s < 0$$

From Algorithm 3, we obtain a decomposition of every tile's flow-out set into MARS. The MARS constructed via Algorithm 3 hold the properties (A), (SP) and (M):

- **Single-producer (SP):** MARS hold the (SP) property by virtue of being subsets of a single tile (which is a parameter).

- **Atomicity (A):** MARS are computed as those iterations that have *exactly* the same consumer tiles: given a set of consumer tiles $I \subset \mathcal{P}_n(\mathcal{T})$, those points consumed by all tiles in I are in set $A(\vec{t})$, and those points not consumed by any tiles outside of I are in $S(\vec{t})$. Intersecting $A(\vec{t})$ and $S(\vec{t})$ gives those iterations that have exactly all tiles of I as consumer tiles.

  Because all points in a MARS share exactly the same consumer tiles, and that input and output are done at the tile level, they have the atomicity property.

Algorithm 3: Computing the MARS

```
Input:  $D$ = iteration space,
        $B = \left\{\vec{b}_i : i = 1, \ldots, Q\right\}$ = PRDG,
        $\mathcal{H} = \{H_i : i = 1, \ldots, t\}$ = tiling hyperplanes
        $\{s_1, \ldots, s_t\}$ = tile sizes
Result: $\mathcal{M}(\vec{t})$ = partition of flow-out into MARS, parametrized by tile coordinates $\vec{t}$
// $T(\vec{t})$ is a tile of coordinates $\vec{t}$
let $T(\vec{t}) = \{\vec{x} \in D : \forall i \in [\![1;t]\!] : t_i s_i \leqslant \vec{x} \cdot \vec{n}_i < (1 + t_i) s_i\}$;
$\mathcal{M} = \varnothing$;
$\mathcal{T} = \mathcal{P}_n(\mathcal{H})$;                         /* All neighboring tiles */
for $I \in \mathcal{P}_n(\mathcal{T})$ do
    $E = \mathcal{T} \backslash I$;
    // $A$: all tiles in $I$ must be reached by $\geqslant 1$ dependence
    for $T \in I$ do
        $N = \mathcal{H} \backslash T$ ;                     /* Hyperplanes not crossed */
        for $\vec{b} \in B$ do
            // $P_{T,\vec{b}}$: $\vec{b}$ crosses all hyperplanes of $T$ and no others
            $P_{T,\vec{b}}(\vec{t}) =$
              $T(\vec{t}) \cap \left\{\vec{x} \in D : \bigwedge_{H \in T} \left(\vec{x} + \vec{b} \text{ crosses } H\right) \wedge \bigwedge_{H \in N} \neg \left(\vec{x} + \vec{b} \text{ crosses } H\right)\right\}$;
        end
    end
    $A(\vec{t}) = \bigcap_{T \in I} \bigcup_{\vec{b} \in B} P_{T,\vec{b}}(\vec{t})$;
    // $S$: no tiles in $E$ may be reached by any dependence
    for $\vec{b} \in B$ do
        // $T$ = a tile that must not be a consumer
        for $T \in E$ do
            $N = \mathcal{H} \backslash T$;
            // $Q_{T,\vec{b}}$: $\vec{b}$ must cross a hyperplane outside of $T$
            // (i.e. in $N$), or not cross all hyperplanes of $T$
            $Q_{T,\vec{b}}(\vec{t}) = T(\vec{t}) \cap$
              $\left\{\vec{x} \in D : \left[\bigvee_{H \in T} \neg \left(\vec{x} + \vec{b} \text{ crosses } H\right) \vee \bigvee_{H \in N} \left(\vec{x} + \vec{b} \text{ crosses } H\right)\right]\right\}$;
        end
    end
    $S(\vec{t}) = \bigcap_{T \in E} \bigcap_{\vec{b} \in B} Q_{T,\vec{b}}(\vec{t})$;
    $M(\vec{t}) = A(\vec{t}) \cap S(\vec{t})$ ;                      /* $M(\vec{t})$ is a MARS */
    $\mathcal{M}(\vec{t}) = \mathcal{M}(\vec{t}) \cup \left\{M(\vec{t})\right\}$;
end
return $\mathcal{M}(\vec{t})$
```

- **Maximality (M):** There is exactly one MARS per set of consumer tiles ($I \in \mathcal{P}_n(\mathcal{T})$). If two points in the flow-out have exactly the same consumer tiles (I), then they belong to the same MARS.

5.3.4 Dual view: flow-in

Equivalently to partitioning the flow-out set into MARS, it is possible to compute the partitioning of the flow-in set of each tile into MARS. The flow-in set is computed with the same algorithm as the flow-out, using the opposite of the dependence vectors.

Intersecting the MARS created by Algorithm 3 (not broken up into individual tile-wise MARS) with the obtained tile-wise flow-in set then gives a breakup into MARS. This partitioning can be used to figure out which MARS every tile should fetch from other tiles as an input.

5.4 Implementation and Analysis

It is possible to express all MARS using polyhedral tools (ISL) provided the tile sizes are constant. However, using the idea from [64], it is possible to use parametric tile sizes by adding those sizes as additional parameters. We have implemented a MARS procedure in Python using ISLPy.

5.4.1 MARS procedure implementation

The MARS procedure is available at `https://github.com/cferr/mars.git`.

To compute the MARS for a given program and tiling hyperplanes, it needs input that can be computed using publicly available tools:

- The polyhedral representation of a program, to be extracted for instance using PET [75];
- Dependence vectors, obtained using array dataflow analysis e.g. using `iscc`;

- Legal tiling hyperplanes, found for instance by calling PLuTo; the standard equation and the normal vectors to these hyperplanes are to be provided.

The MARS procedure then runs Algorithm 3. A visualization of the MARS is given with `islplot` when the iteration space is two- or three-dimensional. For two-dimensional iteration spaces, the MARS in the entire iteration space can be visualized; for three-dimensional spaces, a sample tile needs to be provided and the MARS specific to that tile will be shown.

5.4.2 Handling multi-statement programs

MARS can only be computed for programs exhibiting uniform dependence patterns. A transformation of the iteration space yielded by a polyhedral analyzer may be necessary to obtain such a pattern. This is notably the case with programs where multiple statements exhibit similar dependences.

For instance, we can consider a double-buffered Jacobi 1D code. It is a stencil computation that has an uniform dependence pattern, similar to that of Figure 3.8. The implementation of this stencil found in PolyBench/C[61] is as follows:

```c
static
void kernel_jacobi_1d(int tsteps,
          int n,
          DATA_TYPE POLYBENCH_1D(A,N,n),
          DATA_TYPE POLYBENCH_1D(B,N,n))
{
  int t, i;

#pragma scop
  for (t = 0; t < _PB_TSTEPS; t++)
    {
      for (i = 1; i < _PB_N - 1; i++)
S75_2:  B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
      for (i = 1; i < _PB_N - 1; i++)
S77_2:  A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
```

```
    }
#pragma endscop

}
```

In this code, the computation is done using a double-buffering technique: the same computation is done in two loops over i, alternating which array gets used for reading and writing. Each statement `S75_2` and `S77_2` has its own three-dimensional iteration space, respectively $(t, 0, i)$ and $(t, 1, i)$, and depends on the other statement's results. The dependence between these iteration spaces is not uniform: for instance, to use `A[i-1]` or `B[i-1]`, $(t, 0, i)$ depends on $(t-1, 1, i-1)$ (the dependence is $(-1, 1, -1)$ and $(t, 1, i)$ depends on $(t, 0, i-1)$ (dependence $(0, 1, -1)$).

It is however possible to get a *single* two-dimensional iteration space and a uniform dependence from this program, by interleaving the iteration spaces of the two statements.

Two bijections are necessary to map these two spaces into a single space. The following bijections:

$$T'_0 : \texttt{S75\_2}(t, i) \mapsto (2 \times t, i) \text{ and } T'_1 : \texttt{S77\_2}(t, i) \mapsto (2 \times t + 1, i)$$

give uniform dependences:

$$\begin{aligned}
\mathbf{B} = \{ & B'_0 : (2t, i) \mapsto (2t+1, i); & B'_1 : (2t, i) \mapsto (2t+1, i-1); \\
& B'_2 : (2t, i) \mapsto (2t+1, i+1); & B'_3 : (2t+1, i) \mapsto (2t+2, i); \\
& B'_4 : (2t+1, i) \mapsto (2t+2, i-1); & B'_5 : (2t+1, i) \mapsto (2t+2, i+1)\}
\end{aligned}$$

and it is then possible to combine them into three dependences:

$$\mathbf{B} = \{B''_0 : (t, i) \mapsto (t+1, i); B''_1 : (t, i) \mapsto (t+1, i-1); B''_2 : (t, i) \mapsto (t+1, i+1); \}$$

These three dependences are uniform and apply over the entire (t, i) domain.

Transformations T'_0 and T'_1 change the iteration space in the polyhedral representation. The domain of memory accesses being the iteration space, we must also compose the transformations with these functions to obtain memory accesses from the transformed iteration space. For instance, the two write access functions of this program are:

$$W := \{\texttt{S75\_2}(t, i) \mapsto B[i] \text{ and } \texttt{S77\_2}(t, i) \mapsto A[i]\}$$

We compose them with the inverse of the transformation T'_0 and T'_1 to obtain the new access functions:

$$\begin{aligned} W' := \{&(t, i) \in \{2t : t \in [0; \texttt{\_PB\_TSTEPS} - 1]\} \times [0; \texttt{\_PB\_N} - 1] \mapsto B[i]; \\ &(t, i) \in \{2t + 1 : t \in [0; \texttt{\_PB\_TSTEPS} - 1]\} \times [0; \texttt{\_PB\_N} - 1] \mapsto A[i]\} \end{aligned}$$

where odd values of t result in a write to A and even values of t result in a write to B.

In the following evaluation, this procedure is applied to the `jacobi1d` and `jacobi2d` benchmarks for them to yield uniform dependences.

5.4.3 Results

We have run the MARS procedure against a series of uniform dependence benchmarks. This section evaluates the result on the following questions:

- How many MARS are there per tile?
- What is the dimensionality of the result MARS? In particular, how many singleton MARS are there?

Evaluated applications

The MARS procedure has been used on the following applications:

- `sw`: Smith-Waterman dynamic programming algorithm for sequence alignment;
- `jacobi-1d`: Jacobi 1D stencil;

Table 5.1: Results obtained from the MARS procedure. # CT = number of consumer tiles; # MARS = number of MARS; # Singl. = number of MARS of cardinality 1

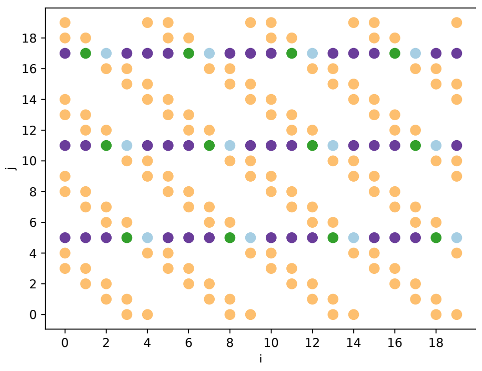
Dims	Application	Dependences	Tiling hyperplanes	# CT	# MARS	# Singl.
2	`sw`	$(1,0), (0,1), (1,1)$	$i+j$, j	3	4	2
2	`jacobi-1d`	$(1,-1), (1,0), (1,1)$	$t+i$, $t-i$	3	4	2
3	`canonical-3d`	$(1,0,0)$, $(0,1,0)$, $(0,0,1)$	i, j, k	3	7	1
3	`gemm`	$(0,1,0)$	i, j, k	1	1	0
3	`seidel-2d`	$(0,1,1)$, $(0,0,1)$, $(1,-1,1)$, $(0,1,0)$, $(1,0,0)$, $(1,-1,0)$, $(0,1,-1)$, $(1,0,-1)$, $(1,-1,-1)$	t, $t+i$, $4t+2i+j$	7	13	2
3	`jacobi-2d-r`	$(1,0,1)$, $(1,1,0)$, $(1,0,0)$, $(1,-1,0)$, $(1,0,-1)$	t, $t+i$, $t+j$	7	13	4
3	`jacobi-2d-d`	$(1,0,1)$, $(1,1,0)$, $(1,0,0)$, $(1,-1,0)$, $(1,0,-1)$	$t+i$, $t+j$, $t-i$, $t-j$	15	34	6

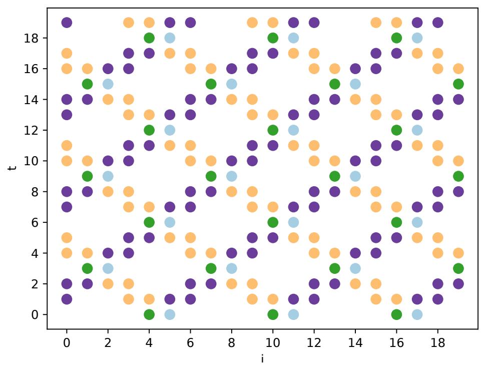
- `canonical-3d`: Artificial 3-dimensional example that has a dependence along each canonical axis;
- `gemm`: GEMM (BLAS) implementation from PolyBench [61];
- `seidel-2d`: Seidel 2D stencil implementation from PolyBench;
- `jacobi-2d`: Jacobi 2D stencil implementation from PolyBench;

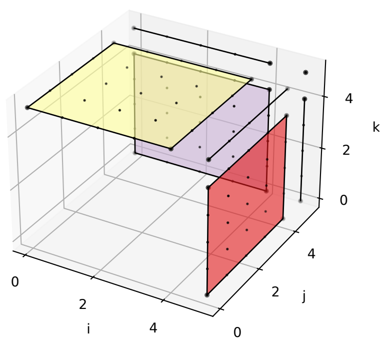
The `jacobi-2d` benchmark is exploited twice, with different tiling schemes: one is rectangular tiling combined with skewing, the other one is diamond tiling [9]. We will refer to them respectively as `jacobi-2d-r` and `jacobi-2d-d`.

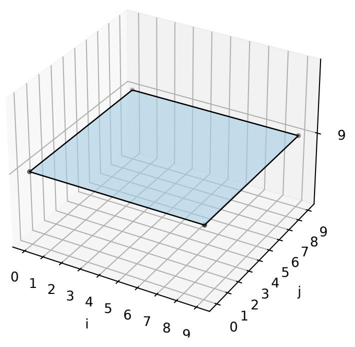
Table 5.1 shows the results obtained by running the MARS computer on the selected applications, and Figure 5.5 shows the MARS for a single tile of every application.

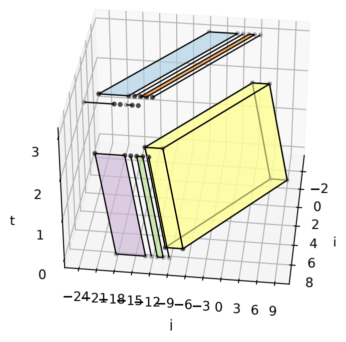
Table 5.1 gives the number of dimensions of the iteration space, the dependence pattern, tiling hyperplanes, the number of consumer tiles per tile (# *Cons. Tiles*), the number of

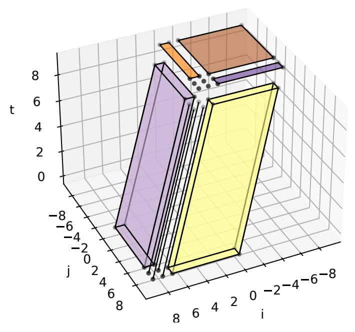

(a) `sw`


(b) `jacobi-1d`


(c) `canonical-3d`


(d) `gemm`


(e) `seidel-2d`


(f) `jacobi-2d-r` (skewing + rectangular tiling)

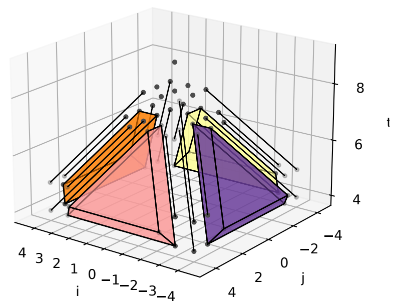

(g) `jacobi-2d-d` (diamond tiling)

Figure 5.5: Visualization of MARS generated using the MARS calcuiator.

Table 5.2: Computation time for MARS over 10 runs

Benchmark	Min (s)	Max (s)	Average (s)	Std. deviation
`canonical-3d`	0.150	0.170	0.163	0.007
`gemm`	0.120	0.130	0.126	0.005
`jacobi-1d`	0.150	0.180	0.162	0.012
`jacobi-2d-d`	11.110	12.040	11.547	0.296
`jacobi-2d-r`	1.130	1.220	1.172	0.031
`seidel-2d`	2.020	2.250	2.100	0.065
`sw`	0.130	0.170	0.154	0.014

computed MARS (*Nb MARS*) and the number of singleton MARS (that have a single point per tile).

For the `jacobi-2d-d` instance, the tiling hyperplanes are not linearly independent. There are indeed three possible tile shapes; despite 34 MARS being determined, only 26 appear in the tile shown in Figure 5.5g.

Analysis

Two observations can be made out of the MARS, on their number and the tiles that consume them. As a general rule, the consumers tiles of a MARS are adjacent to each other, and the more cutting hyperplanes surrounding a MARS, the fewer dimensions it has. One notable case is `seidel-2d` (Figure 5.5e) where a two-dimensional MARS is surrounded by two one-dimensional ones, close to the $t + i$ and $4t + 2i + j$ hyperplanes intersection, and close to the t and $4t + 2i + j$ intersection.

In 2-dimensional iteration spaces, two tiling hyperplanes are enough to tile all dimensions, in which case there are a maximum of $2^{2^2-1} - 1 = 7$ MARS per tile. Our examples, `sw` and `jacobi-1d`, only exhibit 4 MARS, each MARS being consumed by two adjacent tiles.

In 3-dimensional iteration spaces, the number of MARS goes up to 34 per tile on `jacobi-2d-d` out of a maximum 32767 (due to the 15 consumer tiles).

As one can expect given the size of $\mathcal{P}_n(\mathcal{P}_n(\mathcal{H}))$, the complexity of the computation is such that current machines will not find the MARS in a matter of hours if there are 5 or more tiling hyperplanes (i.e. a 2^32-wide space to enumerate). This is a strong call to prune the search space. We implemented several optimizations to alleviate this time:

- Figuring out the actual consumer tiles instead of enumerating $\mathcal{P}_n(\mathcal{P}_n(\mathcal{H}))$. To do this, we only enumerate $\mathcal{P}_n(\mathcal{H})$, and determine for each neighboring tile, it is a consumer tile (i.e. a dependence leads to it). Let $\mathcal{C}$ be the subset of $\mathcal{P}_n(\mathcal{H})$ with the actual consumer tiles.

- Incrementally building the sets of consumer tiles, instead of browsing $\mathcal{P}_n(\mathcal{C})$. If A from Algorithm 3 turns out to be empty, this means that no iteration has the current set of consumer tiles, so we do not need to check for MARS with additional consumer tiles.

The resulting run times are shown in Table 5.2; even a three-dimensional benchmark with four non-linearly independent tiling hyperplanes like `jacobi-2d-d` takes between 11 and 12 seconds to be computed.

5.5 Applications and Future Directions

MARS can be used in a variety of applications where fine-grain knowledge of the tile's flow-in origin and flow-out destination is known. In this chapter, we detail two applications: compression, and memory allocation.

5.5.1 Memory allocation, Compression

MARS can be used to construct a memory allocation for inter-tile communication, similarly to what Zhao et al. [90] have done with coarser-grain blocks. The idea is similar: allocate contiguous blocks of memory for each MARS, and find a suitable layout.

The fact that MARS are not redundant makes them suitable for compression: in general, decompression of an entire block of data is needed to access part of it. When using MARS, all the data that is decompressed is actually needed, and therefore there is no compression-induced redundancy. Such a framework would be similar to works such as Ozturk et al. [58] with data tiles of different sizes, where each data tile is actually a MARS.

5.5.2 Error detection

A well-known technique to make systems fault-tolerant is the use of checksums over data to find out whether errors are present. Techniques such as Algorithm-Based Fault Tolerance [39] can check the result of computations using an accumulation operator over a fraction of the iteration space where the result must match some otherwise computed value. If both checksums do not match, then the results are to be recomputed. MARS can be used to check the integrity of *transmitted* data with the same idea: a checksum transmitted along with each MARS is matched against a real-time checksum. If both do not match, it may indicate a transmission error, or a storage error if the data is in memory.

5.5.3 A case for merging MARS

In some cases, as it can be observed in Figure 5.5, the irredundancy property yields singletons, that may cause performance drawbacks, for instance when creating access functions at the granularity of MARS: such access functions will read or write the exact access data for each tile, but unless singleton accesses are merged or coalesced with other accesses, these will incur a bandwidth waste. Coalescing accesses is done in Chapter 6 for a memory allocation.

Another way to alleviate performance issues is to relax the irredundancy property, and allow for MARS to be merged. This merge process would yield an intermediate partitioning between very fine-grain MARS and the entire flow-out, which would be the result of merging all MARS together; studying the potential benefits of such a merging is left for future work.

5.6 Conclusion

In this chapter, we have introduced an element of program analysis, MARS, to determine sets of data communicated between tiles without redundancy. These sets can be computed for certain programs with uniform dependence patterns, and their computation can be automated.

In the next chapter, we will cover an application of MARS to memory allocation by using their irredundancy. We have identified several other routes from this chapter. The possible use cases of MARS include error detection and correction: thanks to the single-producer property, producer tiles to be replayed can be uniquely identified, while the atomicity and maximality properties yield coarser sets to apply checksum on than singletons. On the computation side, MARS computed in this chapter are limited to uniform dependences, yet a number of common applications feature affine dependences for their inputs (e.g. `gemm`), or between computations (e.g. optimal string parenthesization). Supporting affine dependences warrants a further study, which is done in Chapter 7.

Chapter 6

An Irredundant and Compressed Data Layout to Optimize Bandwidth Utilization of FPGA Accelerators

FPGA accelerators may be used to perform computations on arbitrary data types; their strength is their ability to make custom operators to process data. Working with these data types can be done at a certain cost with respect to memory accesses, because of a padding requirement to keep data aligned at certain boundaries. Such alignment causes redundancy and under-utilization of bandwidth, but is necessary to keep a random access ability to data.

In Chapter 5, we introduce a decomposition of the flow-in and flow-out of tiles of polyhedral programs; this decomposition can be used to transfer data irredundantly, in the sense that we can guarantee a single transfer of each data. However, we must also ensure that all bits of data transferred are actually useful, which padding data is not.

Random access patterns can be avoided by using certain data layouts and access patterns such as CFA introduced in Chapter 4. Keeping the same idea as CFA of using burst transfers, we suggest to build a memory allocation that is burst-friendly, and irredundant in both storage and transfer.

In this chapter, we introduce a data layout such that contiguous access patterns are possible with as little transfer redundancy as possible, using the MARS sets introduced in Chapter 5, and notably their atomicity and irredundancy properties. By exploiting both properties, the data layout we build is contiguous and irredundant, but also a candidate for compression, leading to high bandwidth utilization and data savings.

This chapter makes the following contributions:

- A memory layout using the MARS from Chapter 5 optimized for a maximum of coalescing opportunities,
- An automatic compression and packing scheme for the MARS-based layout,
- An implementation on a selection of FPGA accelerators for polyhedral benchmarks.

It is organized as follows: Section 6.1 gives the motivation of this chapter, and Section 6.2 places this chapter in its scientific context. Section 6.3 explains how the allocation is constructed by laying out MARS and using data packing. Then, Section 6.4 details our FPGA-specific implementation of this allocation, and Section 6.5 evaluates and discusses it.

6.1 Motivation

Leveraging contiguous accesses often comes with a redundancy cost: reading extra values that are not used later on is sometimes more profitable than breaking a burst access to avoid these unused inputs. The cost of such a manipulation can be signficant: considering tiled stencil computations with time skewing (e.g. that from Intel) for stencils, a bounding box of the results can be used. The higher the tiles (and the reuse), the higher the volume of unused data.

Using data flow information to extract contiguity can keep the volume of unused data within controlled bounds, which was shown in Chapter 4. However, this is not sufficient to achieve data compression: because compressed data has a size a priori unknown, positioning data in memory and seeking to random positions, as is still necessary with CFA, is impossible. We therefore need a finer decomposition of the data flow to enable its compressibility while maintaining its contiguity.

6.1.1 Illustrative example: 1D Jacobi stencil

To illustrate the flow proposed in this chapter, we propose a Jacobi-1D stencil as running example. This kernel updates a one-dimensional sequence of values, and computes each point as a weighted average of it and its neighbors:

$$c_{t+1,i} = \frac{1}{3}\left(c_{t,i-1} + c_{t,i} + c_{t,i+1}\right)$$

A C implementation of this stencil is provided in the PolyBench/C suite as the following code:

```c
for (t = 0; t < _PB_TSTEPS; t++) {
    for (i = 1; i < _PB_N - 1; i++)
        B[i] = 0.33 * (A[i-1] + A[i] + A[i+1]);
    for (i = 1; i < _PB_N - 1; i++)
        A[i] = 0.33 * (B[i-1] + B[i] + B[i+1]);
}
```

This stencil operates over a two-dimensional iteration domain *time* × *space* where each point has a coordinate (t, i). Because such a domain may be arbitrarily large, the whole dataset may not fit into FPGA on-chip memory, and needs to be optimized before it can be mapped to the FPGA. This naive implementation of Jacobi-1D cannot fit on-chip for gigabyte-scale problem sizes, thus requiring tiling for locality. For the sake of simplicity of the illustration, we have chosen small, diamond-shaped tiles, illustrated in Figure 6.1.

An accelerator for the tiled program processes the domain tile by tile. To execute a tile, it needs to retrieve intermediate results from previously executed tiles. These intermediate results are located outside of the accelerator, in off-chip memory, and need to be copied into on-chip memory.

For this tiling scheme, intermediate results to be retrieved come from the tiles located below the tile to execute; in Figure 6.2, these tiles are designated as the source of incoming arrows into the tile to execute. Likewise, the outgoing arrows show those intermediate results

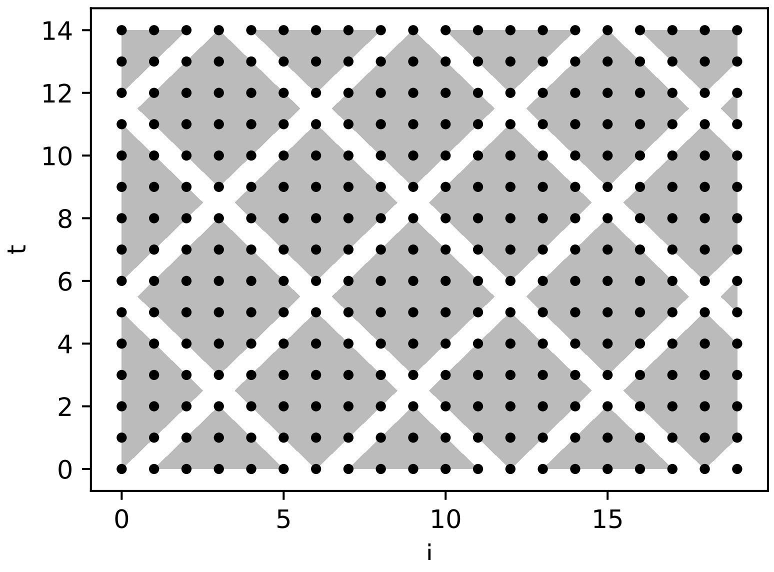


Figure 6.1: Domain of the Jacobi stencil divided into tiles of size 6×6. Each tile contains 18 (t, i) points corresponding to 18 computations of $c_{t,i}$s.

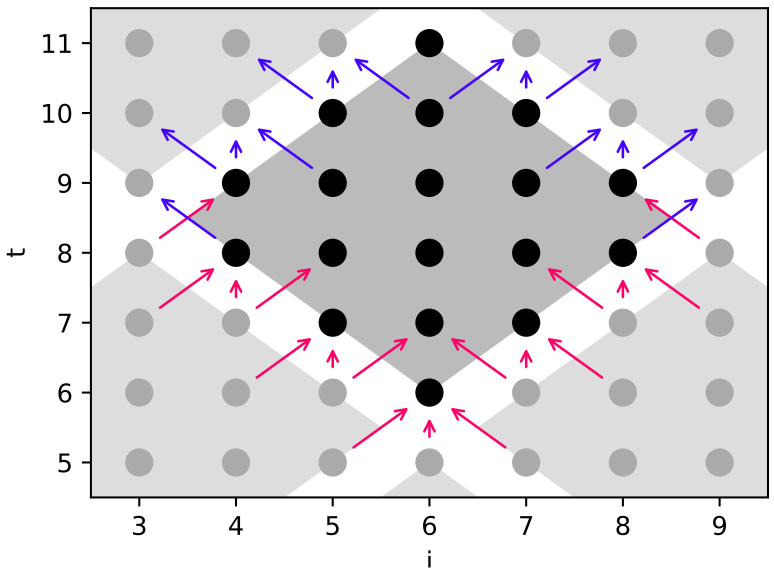


Figure 6.2: Inter-tile communication pattern for the Jacobi stencil: red arrows indicate data input into the tile shown in the center, and blue arrows indicate data output from this tile.

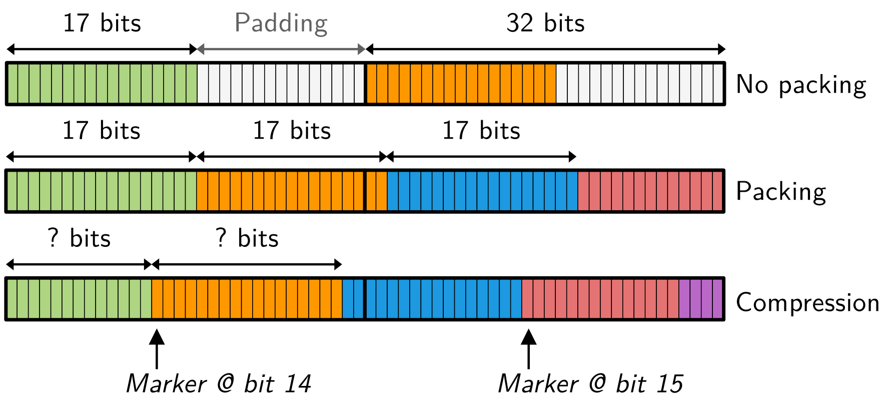


Figure 6.3: Data packing and compression reduce storage and transfer redundancy at the expense of address alignment and, for compression, predictability of addresses.

that will be used by other neighboring tiles. All of these data transfers are the ones this work seeks to optimize; improvements of the compute engine fall out of the scope of this dissertation.

6.1.2 Padding, packing and compression

In order to maximize the utilization of bandwidth, every bit of data transmitted must be useful. However, with domain-specific data types (e.g., custom fixed point), unused bits must usually be transmitted due to memory alignment requirements. In the following, we explain how data contiguity can be leveraged to two ways: packing data to reduce the unused bits transmitted; and compressing data to further save bandwidth.

Data packing

Most memories are byte-addressable and most processor architectures also require aligned accesses at word boundaries, usually at 32 or 64 bits. Although FPGA accelerators can operate on arbitrary-precision data types, off-chip data transfers must abide by the addressing requirements of the external memory. They therefore need to pad the incoming and outgoing data: in practice, for a 17-bit access, 32 bits of data will be transferred, 15 bits of them being wasted in padding.

Note that padding is necessary to enable random accesses to data: it provides the guarantee that a given memory cell contains only the requested data and no manipulation needs to be done to extract it. Data accessed in a contiguous manner does not need this guarantee and may overlap multiple adjacent cells, as simple wire manipulations on the FPGA will give back the original data.

Data packing, as illustrated in Figure 6.3, consists in avoiding padding the data so that words are adjacent at the bit level in memory. Figure 6.3 shows buffer structure for unpacked and packed data of 17 bits in 32-bit words. Unpacked data has aligned addresses, but requires extra storage and transfers unused data; packed data has unaligned addresses but saves storage and avoids some redundant bits from being transmitted. It becomes however impossible to randomly seek in a packed stream due to misalignment without additional data processing, but by definition such random seeks do not happen with contiguous accesses.

In our approach, we leverage contiguous accesses to (i) avoid the adverse effects of packing induced misalignment and (ii) to maximize bandwidth utilization by not padding data.

Runtime data compression

Packing data saves bandwidth by eliminating the padding bits, and is applied independently of the data itself. However, further optimisation is possible by exploiting properties of the data (e.g., correlation between integers in an array) for compression. When this technique is applied right before / after off-chip communications, the design benefits from a reduction of I/O cycles (as the amount of data transferred is reduced) without increase of the computation subsystem as the latency of the compression module can be hidden by the pipelining structure

Compression is easy to apply to contiguous streams of data, but is not to data where indexed or random accesses are necessary. We must exhibit access contiguity, as it is in general impossible to seek within a compressed block without decompressing more data than needed. Figure 6.3 shows that the position of data within a compressed block is unpredictable.

Our approach performs runtime compression and, to maximize its efficiency, creates data blocks with a contiguous access guarantee to ensure every decompressed piece of data is used.

In general, the compression algorithm is domain-specific, e.g., ADPCM for voice [21] or JPEG for images [70]. For FPGA implementations, the choice of the algorithm is also driven by its throughput: compression and decompression must be able to sustain the input and output throughput not to become the bottleneck. We choose to illustrate the idea with a simple differential compression algorithm which encodes a sequence $w_0 w_1 \dots w_n$ of N-bit words as follows:

- Encode w_0 as is.
- For $1 \leqslant i \leqslant n$:
    1. Compute $\Delta = w_i - w_{i-1}$,
    2. Let L be the number of leading zeroes of Δ if $\Delta \geqslant 0$, or leading ones if $\Delta < 0$,
    3. Encode $N - L$ using $\lfloor 1 + \log_2(N) \rfloor$ bits, followed by the sign bit of Δ,
    4. Encode the $N - (L + 1)$ lowest bits of Δ.

This technique is especially effective when the distribution of the transferred data is not spread, typically on benchmarks based on the computation of the average such as our Jacobi-1D example from subsection 6.1.1.

6.1.3 Contributions

The goal of this work is to propose a **source level compiler optimisation to (i) reorganize data in memory to enable contiguous burst access and (ii) further improve bandwidth utilization through packing and compression.** Our optimization pass is meant to be integrated within an HLS polyhedral compilation flow, as illustrated in Figure 6.4; aiming at sitting between the locality optimization phase (tiling) and the HLS

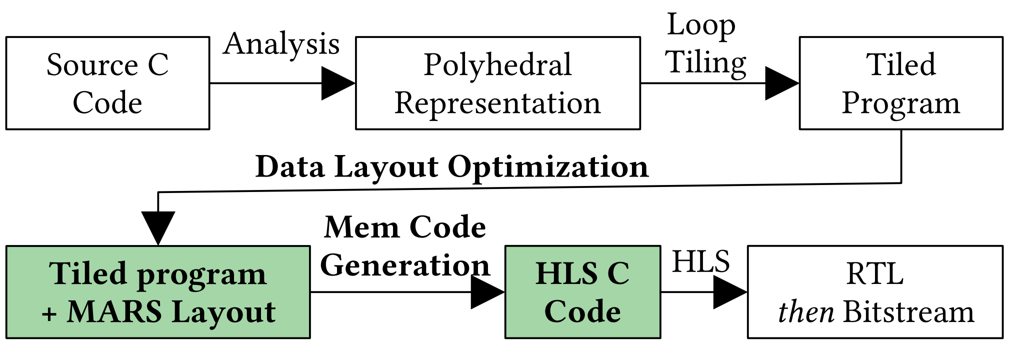


Figure 6.4: Compiler flow (contributions of this chapter in green)

synthesis stage. In fact, **our approach does not replace locality optimizations**, it **complements them**.

6.2 Related Work

This work comes as part of a global effort to relieve memory-boundness of high-performance accelerators. In this section, we study other techniques used to relieve the memory wall, some of which may not apply to compilers due to not being automatable or breaking program semantics.

6.2.1 Data Compression

Data compression saves bandwidth without requiring to modify the program's algorithm. It is therefore suitable for many bandwidth-bound problems.

Compression techniques

Data compression in FPGA accelerators is already a necessity for some intrinsically memory-bound applications such as deep convolutional networks, as no locality optimization can bring further bandwidth savings. We here focus on two kinds of compression: lossless and lossy.

Lossless compression Lossless compression guarantees that the decompressed data is exactly the same as the data before it was compressed. This property makes it possible to do seamless, inline compression and decompression as is done for MARS. This is commonly performed in deep neural network accelerators [1, 34]

Sparse encoding can be considered a form of lossless compression, and is also commonly found in machine learning applications [26, 48]. Sparse data structures often require indirections, which make them unsuitable for use in polyhedral compiler flows unless the sparse structure is immutable [38].

Lossy compression It is possible to save more storage and bandwidth by using lossy compression. Some applications in machine learning can afford a loss of precision without degrading the quality of the result, e.g. using JPEG-compressed images [57] as inputs. However, lossy compression alters the data and cannot be automatically inserted by a compiler unless the user explicitly requests it.

Dynamic data compression

In this work, we automate the compression and decompression of data and it is transparent to the computation engine on FPGA. Other works [58, 67] perform dynamic, demand-driven compression without prior knowledge of the data to be handled. Thanks to the static control flow of polyhedral codes, all the data flow is statically known and it is not necessary to maintain a cache policy.

6.2.2 Memory access optimization

The layout we propose in this work optimizes memory accesses by exhibiting contiguity using polyhedral analysis. In this section, we go through other polyhedral memory access optimizations, and explain other non-polyhedral ways it is possible to improve memory accesses.

Polyhedral-based optimizations

Using the polyhedral model and loop tiling to capture the data flow is the subject of a number of works, proposing different breakups of the dataflow. Datharthri et al. [23] and

Bondhugula [7] propose decompositions of the inter-tile communications to minimze MPI communications. This work also seeks to optimize the passing of intermediate results, but the data allocation is not statically determined like in this work.

A MARS-like decomposition of the inter-tile data flow into coarse-grain blocks for MPI has been proposed by Zhao et al. [90]; our work achieves irredundancy which requires a finer-grain modeling than the one proposed by [90].

Domain-specific optimizations

Memory access optimizations such as a change of data layout or access pattern can also be specific to each problem. We show here two cases of domain-specific optimizations.

Data blocking Data blocking (or tiling) is memory layout transformation that chunks multi-dimensional arrays into contiguous blocks. Similar to loop tiling, data blocking allows to coalesce accesses to entire regions of the input or output data.

Data blocking can be efficient when the memory footprint of one iteration of an accelerator corresponds to a data tile. Although it has been used to optimize machine learning accelerators [73], it may break spatial locality and degrade performance of accesses that cross tile boundaries.

Data blocking can be combined with loop tiling and polyhedral analysis to coalesce inter-tile accesses. Ferry et al. [31] seeks to exhibit the largest possible contiguous units spanning multiple tiles.

Stencil optimization Stencil computations have regular and statically known memory access patterns. Domain-specific optimizers like SODA [13] derive an optimized FPGA architecture and memory layout specific to each stencil.

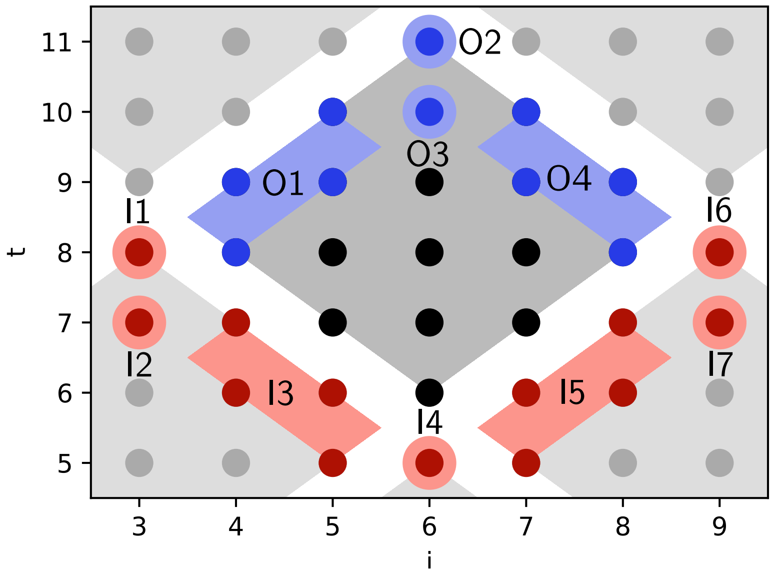


Figure 6.5: MARS: Groups of points within a tile which data is contiguous in global memory. In blue, the MARS produced by the center tile (O1 to O4); in red, the MARS consumed by that same tile (I1 to I7).

6.3 Memory allocation for FPGA accelerator

This section discusses the memory allocation we propose using the MARS: first, it formulates how to lay out MARS in memory to maximize contiguity of accesses at readback time; then, it explains how data packing is dealt with using MARS.

6.3.1 Extracting Contiguous Data Blocks

The first step in our method consists in analyzing a program's behavior with respect to memory, to determine which data can/should be grouped together as contiguous blocks. The sought groups of data honor two properties:

- Atomicity: If any data in the group is needed for an instance of the accelerator's execution flow (a tile), then the entire group is also needed for the same tile.
- Irredundancy: No data is retrieved or stored more than once into memory throughout the execution of a single tile.

These groups of data are determined by using the analysis technique from Chapter 5 within a polyhedral compiler. This analysis yields sets of on-chip memory addresses, such that all the data from these on-chip cells will be allocated a contiguous block of data in off-chip memory.

Example

Applying the MARS analysis from Chapter 5 to the Jacobi stencil of Section 6.1.1 gives the sets of addresses corresponding to the points illustrated in Figure 6.5:

- For the input of each tile, seven contiguous blocks of data labeled I1 to I7 are to be taken, across three different producer tiles.
- For the output, each tile will produce four contiguous blocks of data labeled O1 to O4.

There is a correspondence between output blocks (MARS) Ox from a tile and input blocks Iy from other tiles: each Ox corresponds to one Iy in several other tiles.

Without any further information, the result of MARS analysis would make the accelerator require seven input and four output burst accesses. This number could potentially be reduced. If I1, I2 and I3 were adjacent in memory, it would be possible to make a single access instead of three, and likewise for I5, I6 and I7. The total number of input accesses would go down to just three.

In order to reduce the number of accesses to the above, we have to show that it is actually achievable: the blocks I1 through I7 are read by multiple tiles, and coalescing opportunities for one tile may be incompatible with another tile's coalescing opportunities.

The next subsection formalizes this example into an optimization problem seeking to minimize the number of accesses.

6.3.2 Enabling Coalesced Accesses across Contiguous Data Blocks

From the polyhedral analysis of the previous subsection, we have determined sets of on-chip data to be grouped as contiguous blocks of data, called MARS. How these blocks are laid out in memory is important for access performance: if multiple MARS happen to be accessed in a row and they are adjacent in memory, the accesses to these MARS can be coalesced into a single access and better utilize bandwidth.

This section explains how the "outer layout" of the MARS is determined so as to maximize the coalescing opportunities.

Properties of the layout

The goal of this work is to minimize the number of I/O cycles, and therefore the data layout must exhibit contiguity (for both reading and writing). However, that contiguity must not come at the price of an increase in I/O volume. To model this constraint, we apply two hypotheses.

Contiguous tile-level allocation We are looking for a layout of MARS in memory, and know that compression will be applied to them. Due to the size and position of compressed blocks being unpredictable, it is not feasible to interleave MARS from multiple tiles in memory. Therefore, **we allocate each tile a contiguous block of memory for its MARS output**.

This allocation has two consequences: the write side can be done entirely contiguously, and we only have to optimize contiguity at the read side.

Irredundancy of storage Under the previous hypothesis, we want to maximize the coalescing opportunities between MARS accesses for the read side only. While it is possible to obtain this contiguity by replicating the MARS in multiple layouts, one per consumer, doing so would defeat the goal to save I/O cycles. We therefore choose to **store each MARS only once in memory** (irredundant storage).

The goal is now to find a single layout for the MARS produced by each tile, that exhibits as much read-side coalescing opportunities as possible. We obtain it through an optimization problem that is defined in the next subsections.

Example

In the example of Section 6.3.1, it appeared that the number of burst accesses could go from 7 to 3. Let us show there actually exists a layout achieving these 3 bursts.

Figure 6.5 shows the correspondence between input and output MARS:

- I1, I2 and I3 come from the southwest tile, corresponding to its O2, O3 and O4 blocks. We would like these three MARS to be contiguous, regardless of which relative order, to make a single burst.
- I4 comes from the south tile, corresponding to its O2 block.
- I5, I6 and I7 come from the southeast tile, corresponding to its O1, O2 and O3 blocks. We would also like them to be contiguous.

We do not make any hypothesis on the relative location of data from the southwest tile, the south tile and the southeast tile. This makes it impossible to obtain fewer than 3 burst accesses.

The information we have at this point can be used as the constraints and objective of an optimization problem: we want to maximize the number of contiguities in the layout among those desired, under the irredundancy constraint. We provide a solver with the following problem:

- Maximize the contiguities among the desired ones: make MARS O2, O3 and O4 contiguous in any order, and make MARS O1, O2 and O3 also contiguous in any order.
- Per the hypothesis of Section 6.3.2, we want a layout of MARS O1, O2, O3 and O4.
- There can be no fewer read bursts than 3.

The solver returns the following layout of the output MARS for each tile: O1, O3, O2, O4.

Looking from the consumers, I1, I2 and I3 (resp. southwest O2, O3 and O4) are contiguous; I5, I6 and I7 (southeast O1, O2 and O3) are also contiguous. We can therefore coalesce, for each tile, the reads I1, I2 and I3 into a single burst, and I5, I6 and I7 into another burst, achieving the three sought input bursts.

General case

In this section, we lay out the blocks of data from the MARS analysis to maximize the coalescing opportunities between them.

With the allocation choice of Sec. 6.3.2, writes are guaranteed to be done without discontiguity. We therefore lay out the MARS to make the *read* side as contiguous as possible. In other words, we need to lay out the MARS in memory so that as many MARS as possible can be read as a *coalesced* burst.

We propose to model this problem as an Integer Linear Programming optimization problem as described in this section. Intuitively, if a pair of MARS is needed by a consumer tile and the two MARS are next to each other in memory, then a coalesced access for the two (a "contiguity") is issued. We therefore seek to maximize the number of such contiguities.

The solution to this optimization problem, given by the solver is an ordered list of the MARS produced by each tile, that allows the minimal number of transactions to read all MARS input of a tile.

The problem is modeled in two cases: first, the case where all tiles have a single shape and emit the same MARS; and second, the case where multiple tile shapes exist, in which case, as stated in Chapter 5, each tile shape emits different MARS and multiple layouts need to be derived.

Encoding data structures for a single tile shape

We need a number of intermediate variables in order to encode the layout problem as an ILP problem. We are going to use two data structures to encode the layout permutation: an explicit permutation mapping, and a successor matrix.

Consider M MARS; let each MARS be assigned a number from 0 to $M - 1$. The permutation is then represented by:

- $\delta_{i,j} \in \{0, 1\}$ for successor variables; $\delta_{i,j} = 1$ encodes that MARS i is immediately before MARS j in memory.
- $\gamma_i \in \{0, \ldots, M-1\}$ for the explicit permutation; γ_i is the position where MARS i will be in the final layout.

The successor matrix encodes the fact that no MARS can have multiple predecessors or successors with the following constraints:

- $\forall i : \delta_{i,i} = 0$
- $\forall i : \sum_j \delta_{i,j} \leqslant 1$ (a MARS can only precede at most a single other one)
- $\forall j : \sum_i \delta_{i,j} \leqslant 1$ (a MARS can only follow at most a single other one)
- $\sum_i \sum_j \delta_{i,j} = M - 1$ (because there are M MARS, we need exactly $M - 1$ successor relations)

The permutation σ is such that $\sigma(i)$ is the new position of MARS number i. σ is represented by γ_is where:

- $\forall i : 0 \leqslant \gamma_i \leqslant M - 1$
- $\forall i, j : (\gamma_j - \gamma_i = 1) \Leftrightarrow (\delta_{i,j} = 1)$ (link between the permutation and successor matrix), which is encoded using an indicator function as $\forall i, j$ s.t. $j \neq i : \delta_{i,j} = \mathbb{1}_{\gamma_j - \gamma_i = 1}$
- $\forall i, j : (i \neq j) \Rightarrow |\gamma_i - \gamma_j| \geqslant 1$ using Gurobi's "absolute value" function.

All these constraints are added to the LP, atop of which additional constraints:

- $\forall i, j : \delta_{i,j} + \delta_{j,i} \leqslant 1$ (only one of MARS i, j can follow the other),
- $\sum_{i=0}^{N-1} \gamma_i = N(N-1)/2$ (the sum of all permutations is the same as the sum of the $N - 1$ first integers)

The objective function to maximize is the **number of contiguities**. It is computed using the successor function and all pairs of MARS that are consumed per producer tile:

$$\max \sum_{P \in \mathcal{P}} \sum_{i=1}^{M} \sum_{\substack{j=1 \\ j \neq i}}^{M} a_{P,i,j} \delta_{i,j}$$

Encoding data structures for multiple tile shapes

For the `jacobi2d-d` benchmark, we observe multiple tile shapes due to the tiling hyperplanes not being linearly independent. Each tile will belong to a **family** of tiles that have the same shape. We'll then take for each family, the MARS that come out of the tile shape this tile family has.

Each family will produce a different number of MARS and consume MARS from other families. Families are given by relations R_i that link tile coordinates. For instance, for `jacobi2d-d`, there are three such relations:

- $R_0 : k4 = k1 - k2 + k3$ (full tiles),
- $R_1 : k4 = -1 + k1 - k2 + k3$,
- $R_2 : k4 = 1 + k1 - k2 + k3$

For a given family R, let $\mathcal{P}(R)$ be the producer tiles this family of tiles is taking data from. Each family produces $M(R)$ MARS. Each input MARS from a producer tile $P \in \mathcal{P}(R)$ is annotated with the relation $R(P)$ ($R(P) \in \{R_0, R_1, R_2\}$).

If $\mathcal{R}$ is the set of family relations, then the LP is modified as follows: for each $R \in \mathcal{R}$, we introduce a permutation σ^R and variables γ^R, δ^R that verify the properties of the above LP definition.

Then, we modify the objective function to maximize the sum of contiguities across all families:

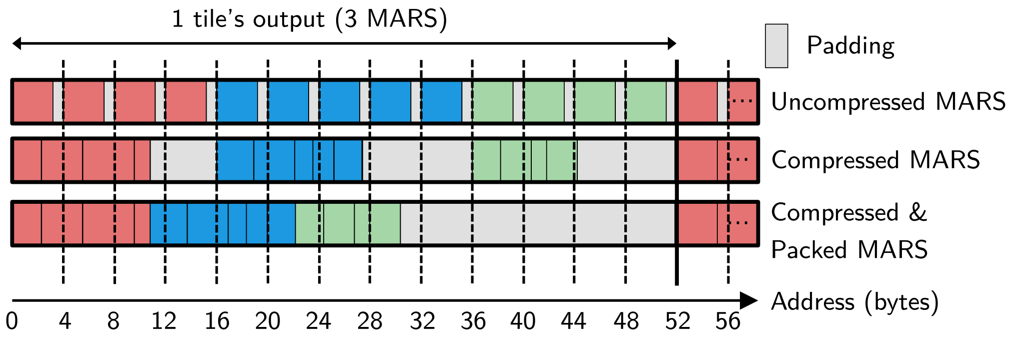


Figure 6.6: MARS data shown without compression, with compression (inside the MARS) and with MARS compression and packing. Packing the compressed MARS preserves the contiguity of coalesced accesses.

$$\max \sum_{R \in \mathcal{R}} \sum_{P \in \mathcal{P}(R)} \sum_{i=1}^{M(R)} \sum_{\substack{j=1 \\ j \neq i}}^{M(R)} a_{P,i,j} \delta_{i,j}^{R(P)}$$

Further work could consider ponderation based on the relative frequency of tile families. Furthermore, for the `jacobi2d-d` benchmark, we execute only full tiles of relation R_0 on the FPGA, and so only these will benefit from the layout; other tiles executed on the host do not take MARS as input.

The layout created in this section honors the *irredundancy* property of the MARS (see Section 6.3.1), but does not yet take full advantage of their *atomicity*: the fact that the MARS are contiguous blocks of data makes them ideal candidates for data packing and compression. This is what we perform in the next subsection.

6.3.3 Contiguity-Preserving Block Compression

So far, our approach has given a layout of data in memory enabling coalesced accesses to contiguous blocks of data produced and consumed by an accelerator. These blocks have an atomicity property that we can further exploit to save bandwidth, by applying data packing and compression, as illustrated in Figure 6.6.

Combining compression and packing

Compressed blocks of data must be considered atomic in the sense that no random accesses into them are possible. This atomicity property is borne by the MARS, as each MARS data block is entirely used when it is accessed, i.e. there are no partial accesses to a MARS.

Compressing the MARS reduces the size of the data and therefore saves bandwidth and storage space; however, it can also break the contiguity brought by the layout of Section 6.3.2 as illustrated by Figure 6.6. To preserve it, we also apply *packing* to the compressed MARS, making them immediately adjacent to each other in memory. Packing compressed MARS also spares the accelerators from unused reads due to padding.

Need to preserve metadata

As the size of compressed blocks depends on their data, it is impossible to know the exact size of each access. However, the size of a burst access must be known prior to the request being issued; additionally, using an estimation of the size or an over-approximation would result in unused input data or additional requests to fill in missing data.

In order to be able to exactly fetch the right size, it is necessary to keep track of the size of each compressed MARS. Moreover, the packing of compressed MARS means that the start of a compressed block may be improperly aligned. It is also therefore necessary to keep track of the alignment of each MARS for proper decompression. In our implementation, bookkeeping is done using on-chip *markers* that are filled in after each MARS is compressed. Details are in Section 6.4.3.

Packing will cause unused input data to enter; however, its size is bounded to one aligned word at the beginning and one aligned word at the end of each transaction. This input redundancy is notably independent of the size of the MARS.

6.4 Implementation and Analysis

In this section, we show how we transform an HLS accelerator description in order to optimize its off-chip memory accesses for bandwidth utilization.

The off-chip data layout and compression proposed in Section 6.3 can be automatically implemented around the existing description of a tile in HLS. The result is a sequence of steps:

- Read MARS layout data and non-MARS input data from off-chip memory into on-chip FIFOs,
- Decompress the input data into FIFOs,
- Dispatch MARS data into on-chip buffers with an allocation suitable for computation,
- Perform the computations onto on-chip buffers,
- Collect MARS output data from the on-chip buffers into FIFOs,
- Compress the collected data,
- Write back the results into MARS layout in off-chip memory.

This section explains how this implementation has been done using high-level synthesis tools: it covers how the polyhedral control flow is simplified for the FPGA accelerator (Section 6.4.1), how the complex data structures describing the MARS are turned into two simple decompression/dispatch and collect/compression steps (Section 6.4.2), and how compression is implemented (Section 6.4.3).

6.4.1 Simplifcation of polyhedral control flow on FPGA

Besides tiling, we have to perform a number of optimizations to get a working accelerator with MARS input/output without incurring a significant slowdown or increase in accelerator's area.

Tile shifting

It is desirable to have a simple control flow on the accelerated part of the program. One such simplification that is made possible by the uniform dependence is to make on-chip computations independent of the tile's position in the iteration space.

All iterations within a tile can be decomposed into the sum of two vectors: one vector $\vec{A}$ which goes to one point in the tile (called the tile's origin), and one vector $\vec{b}$ from that point to the actual iteration, such that the latter does not depend on the tile's parameters. Thanks to the uniform dependence pattern, iteration $\vec{b}$ has the same dependences as $\vec{A} + \vec{b}$, and therefore we can iterate only over the space of a tile defined by the $\vec{b}$s, which greatly simplifies code generation.

Host/FPGA dispatching of tiles

The FPGA accelerator must have a simple control structure to exhibit as much parallelism as possible. Therefore, only *full tiles* are executed on FPGA. Full tiles also all have the same volume of I/O, regardless of their position in the iteration space.

Partial tiles, i.e. those that contain space boundaries, are run on the host CPU, using the original program's allocation. To permit this, data computed on FPGA is taken back from MARS into the original program's memory, and MARS are created back from partial tiles results. It can be demonstrated that no FPGA tiles need any missing MARS data from partial tiles, and therefore there is no issue in writing part of the MARS for these tiles.

The operations performed to execute a partial tile (on the host) are:

- Read MARS from neighboring full tiles that were executed on FPGA, remap their data to its original location,
- Execute the tile's iterations using the original allocation,
- Write back MARS by copying data from the original allocation, skipping cells that would be in MARS yet have no producer iteration.

The control flow necessary for compression would significantly lengthen the execution of host tiles. Therefore, only tiles which producers and consumers are all executed on FPGA will use compression.

6.4.2 From MARS to Collect/Dispatch Functions

Similarly to computations, the control flow of transfers has to be as simple as possible. This way, the HLS tool can extract the requested burst accesses from the code. However, MARS have complicated shapes as illustrated in Figure 5.5, and cannot in general be described as a single, perfect loop nest without guards.

The computations on the tile are independent of the tile's parameters, therefore the on-chip memory addresses accessed for different tiles will be the same, and the placement of MARS will not vary across tiles. We therefore statically compute the position of MARS data and place it in a ROM.

Read and write operations then happen with a very simple loop nest that takes the on-chip addresses from that ROM while performing sequential accesses on the external bus.

The input and output data of each tile is respectively copied into and out of on-chip buffers before the tile execution takes place and after it has fully completed. This is the step where the data goes from a contiguous layout to a non-contiguous layout (suitable for execution) and vice-versa.

Implementing these dispatch and collect steps requires to describe each MARS so that the data contained in it is placed into, or taken from, the right location in on-chip memory. Before dispatch and after collect, the data is located into FIFOs in the contiguous layout.

MARS can have arbitrary complex shapes, and cannot in general be described using simple loops. However, it is possible to fully unroll these loops and obtain a list of on-chip addresses for each MARS. Such unrolled lists are placed into read-only memories on chip. Iterating through these ROMs as in Figure 6.7 gives the corresponding addresses. The size

```
// MARS Dispatch
for(int i=0; i<33; ++i) {
  // take on-chip address from ROM
  struct mars_transfert mt = FPGA_MARS_IN_TBL[i];
  switch (mt.array) {
    // on-chip random write
    case MARS_DATA_ENUM::A: {
      marsToMem_A(mt.dim0, mt.dim1);
      break;
    }
    case MARS_DATA_ENUM::B: {
      marsToMem_B(mt.dim0, mt.dim1);
      break;
    }
  }
}
```

Figure 6.7: Structure of the MARS dispatch implementation (off-chip to on-chip layout)

of these ROMs is notably only dependent on the tile size, and not on the problem size or data type.

6.4.3 Automatic compression

When the data is in the contiguous layout in the form of MARS, it can be seamlessly compressed and decompressed, and the compressed MARS can be packed to preserve contiguity. We explain here the compression, packing and decompression steps, along with how the compression metadata is taken care of.

Compressing Data and Packing MARS

The compression step is relatively straightforward: the compression module takes its input from the collect step FIFO, and generates a compressed stream of data from it. The layout of the data in this FIFO is not altered by the compression step. Likewise, the decompression step takes a stream of compressed words and decompresses it into a FIFO, which is then used by the MARS dispatch step. MARS packing is transparently implemented by the compression step: because MARS are provided in a contiguous manner from the collect step, the first word of each MARS will be immediately adjacent to the last word of the previous MARS in the compressed data stream.

Our compressor, which algorithm is given in Section 6.1.2, is pipelined with an initiation interval of 1 cycle, despite a loop-carried dependence.

The difficult part to implement is decompression: because not all MARS from a given tile are decompressed, we need to be able to seek at the start of a particular MARS. This ability is given by metadata described in the next paragraph.

Metadata management

The consequence of MARS compression is that their size is unknown a priori. To preserve the contiguity of the layout from Section 6.3.2, we must avoid padding the compressed MARS to preserve alignment, Therefore, the compressed MARS are packed and immediately adjacent to each other in memory.

To keep track of the position of each MARS, we use a data structure with two pieces of information: a *coarse-grain* position indicating how far (in aligned words) to seek, and a *fine-grain* position marker that specifies which bit is the first of the said MARS.

Because the length of MARS is known at compile time and constant across tiles, the position of the markers within the uncompressed stream is also constant. Therefore, like the MARS descriptions, the positions of markers (i.e. start of each MARS) within the uncompressed stream are put into a ROM:

```
#define NB_MARKERS 3
#define MARKERS {62, 63, 64}
```

The markers for the compressed stream are maintained within an on-chip cache, which size is specified at synthesis time via a macro:

```
struct compressed_marker<NB_MARS_POS_BITS, LOG_BUS_WIDTH> markers[
    COMPRESSION_METADATA_SIZE][NB_MARKERS];
```

The allocation within this cache is done from the host: registers are used to specify whether a tile's MARS are compressed, whether its dependences are, and where the markers

Benchmark	Tile Sizes	In MARS	Out MARS	Read Tr.	Write Tr.
`jacobi-1d`	6×6, 64×64, 200×200	7	4	3	1
`jacobi-2d`	$4 \times 5 \times 7$, $10 \times 10 \times 10$	28	13	10	1
`seidel-2d`	$4 \times 10 \times 10$	33	13	10	1

In MARS = Number of flow-in MARS; Out MARS = Number of flow-out MARS;
Read Tr. = Number of (flow-in) read transactions;
Write Tr. = Number of (flow-out) write transactions

Table 6.1: Characteristics of the selected benchmarks. The number of bursts per tile accounts for layout-induced access coalescing and is independent of tile and problem size.

for its dependences are located. This location depends on the size of the space; for the Jacobi stencil, the formula is:

```
unsigned compressionMetadataAllocation(
 int tsteps, int n, int M1, int M2, int k1, int k2) {
  return (k2) + M2DEC_FORMULA + M2 * ((k1 - 1) & 0x01);
}
```

It should be noted that the `markers` structure is persistent between runs. It is updated by the MARS write step and used by the MARS read step. This update prevents the current HLS tools from constructing a macro-pipeline (e.g. using the `HLS DATAFLOW` pragma) unless the structure is in a separate module.

6.5 Evaluation

We evaluate our approach with respect to the following questions:

- **Compile-time performance:** How much time does it take to compute the MARS layout?
- **Design quality:** How does using MARS affect the FPGA accelerator's area consumption?
- **Runtime performance:** How much I/O cycles do compressed MARS save with respect to a non-MARS memory layout?

- **Applicability:** How does the data type, tile size and problem size affect the compression ratio?

6.5.1 Protocol and benchmarks

Benchmarks

We have selected the following applications from the PolyBench/C suite[61]:

- `jacobi-1d`: Jacobi 1D stencil, as used in the running example;
- `jacobi-2d`: Two-dimensional version of the Jacobi stencil, exhibiting few and simple MARS;
- `seidel-2d`: More complex benchmark exhibiting a higher number of MARS with more complex shapes.

Layout determination was done using the Gurobi solver (version 10.0.3 build v10.0.3rc0 (linux64)).

The data types used are fixed-point numbers (18 bits, 24 bits, 28 bits) and floating-point numbers (float, double). We also ran simulations with a 12-bit fixed-point data type without synthesizing it, Vitis HLS being unable to infer bursts from that data type.

The chosen applications provide a non-MARS data layout in their original code. Because FPGA developers usually try to seek burst accesses where possible, we have created two access patterns on the non-MARS layout to compare against that try to exhibit bursts:

- A minimal access pattern, fetching and storing the exact I/O footprint of the tile, letting the HLS tool infer bursts where possible.
- A rectangular bounding box of the accessed data like done in PolyOpt/HLS [60], which description is simple enough to infer only burst accesses.

Both access patterns are generated using a polyhedral code generator available in ISL [74].

Hardware platform

We used a Xilinx ZCU104 evaluation board, equipped with a `xczu7ev` MPSoC. We ran Pynq 3.0.1 with Linux 5.15 and synthesis was done using the Vitis/Vivado suite version 2022.2.2 All benchmarks, are running at a clock frequency of 187 MHz and communicate with the off-chip DDR using one non cache-coherent AXI HP port.

Protocol

Each benchmark is run for each data type, each space size and tile size. Part of the computation is done on the host: incomplete tiles are executed on a single thread on the Cortex-A53 CPU of the MPSoC. Transfer cycles are measured only for the FPGA tiles and do not account for the host.

Cycle measurements are gathered using an on-FPGA counter and the area measurements are extracted from Vivado place and route reports.

Table 6.1 shows the characteristics of each benchmark, in terms of number of MARS, and number of bursts after coalescing optimization of Sec. 6.3.2.

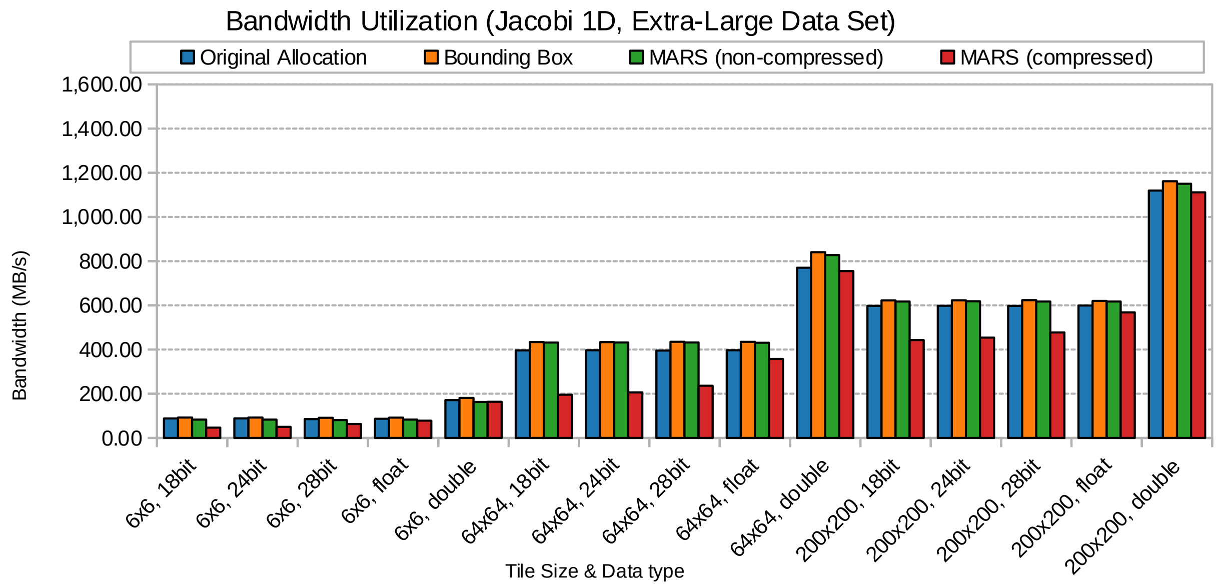
6.5.2 Results and discussion

Runtime performance

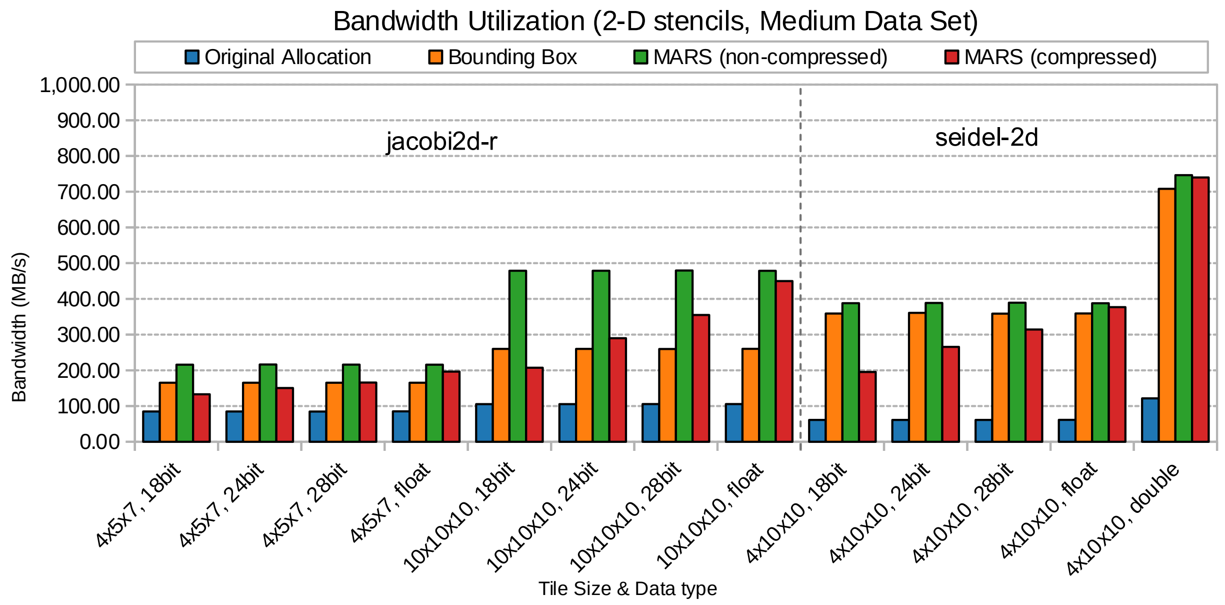
Figure 6.8 shows the bandwidth measured outside the accelerator for each data type and each benchmark; Figure 6.9 shows the *effective* bandwidth, which accounts for the compression ratio (counted positively) and the redundancy due to padding (counted negatively).

It appears that the compressed MARS baseline consistently shows an increase in bandwidth utilization, regardless of the benchmark and baseline. There are multiple factors to explain this increase.

Compression and iteration space dimensionality For the `jacobi1d` benchmark, the original one-dimensional allocation already enables the exhibition of burst accesses. In this case, non-compressed MARS baseline gives little performance increase compared to the

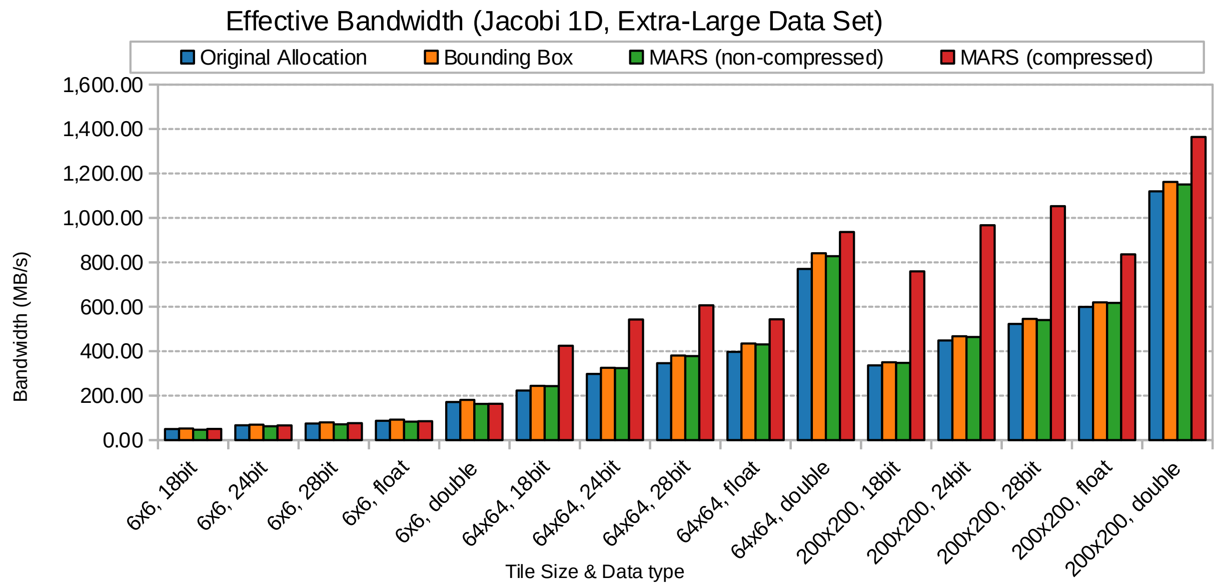


(a) `jacobi-1d`

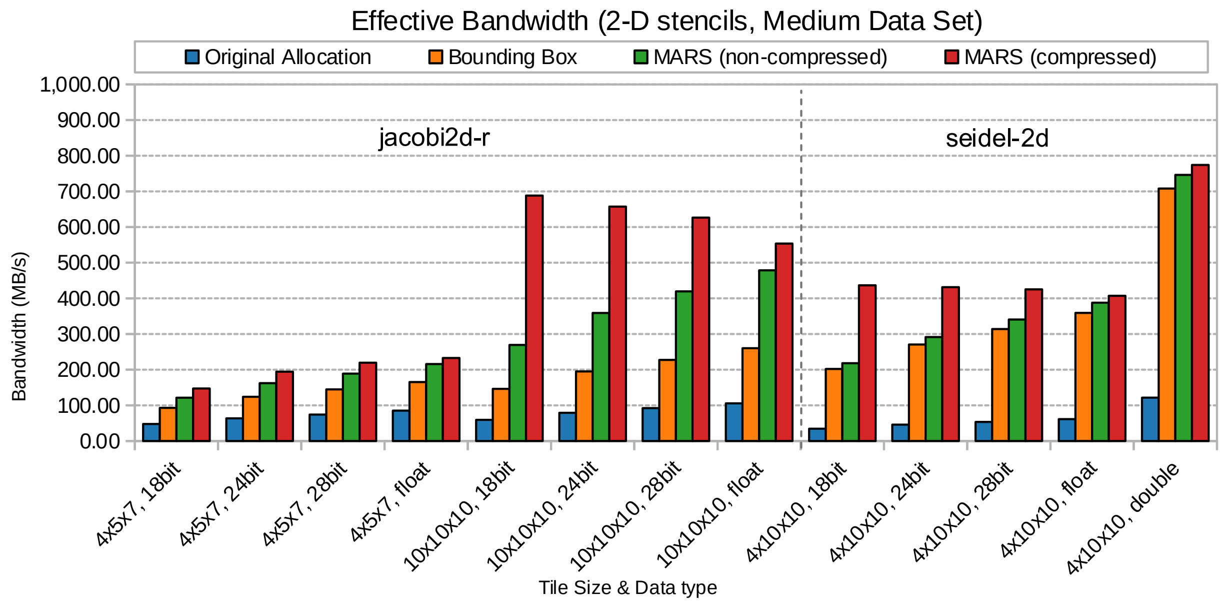


(b) `seidel-2d`, `jacobi-2d`

Figure 6.8: Bandwidth utilization as seen from outside the accelerator.



(a) `jacobi-1d`



(b) `seidel-2d`, `jacobi-2d`

Figure 6.9: Effective bandwidth: bandwidth after decompression, minus redundancy.

orignal allocation and bounding box. However, using compression increases the effective bandwidth, even more so with larger tiles: the larger the tile, the more the padding data transferred. For small tile sizes like 6×6, the gains are marginal if any: the number of compressed elements is too small to exhibit large gains from compression. This effect is also visible in the `jacobi2d` stencil with tile size 10×10 (Figure 6.9b). Compression algorithms selected based on the data used could yield a higher effective bandwidth, yet our benchmarks do not feature real-life data sets.

For the `2d` examples that have three-dimensional iteration spaces, using MARS, even without compression, is already profitable compared to the non-MARS layouts: most of the gains are due to contiguity more than compression.

Effect of data type On the `jacobi-1d` benchmark, the choice of a 200×200 tile size shows a more significant benefit in using compressed MARS for fixed-point data types than floating-point. This is explained with the better compression ratio: when modeling continuous spaces like those used on the Jacobi stencils, neighboring fixed-point values will have more higher bits in common than floating-point data where neighboring values mostly only share the exponent.

Compile-time performance

Table 6.2 shows the time it took for each benchmark to be run through the layout determination and code generation framework. The compilation process does not take more than a few seconds to execute for the benchmarks we selected, starting from the polyhedral representation of the program to the end of HLS code generation. Notably, the layout determination ILP problem only depends on the number of MARS and is independent of the tile size.

Table 6.2: Layout Computation and Code Generation Time

Benchmark	Tile Size	Compile Time (s)
`jacobi-1d`	6×6	0.76
`jacobi-1d`	64×64	0.68
`jacobi-1d`	200×200	1.02
`jacobi-2d`	$4 \times 5 \times 7$	5.57
`jacobi-2d`	$10 \times 10 \times 10$	5.09
`seidel-2d`	$4 \times 10 \times 10$	3.21

Design quality

Figure 6.10 shows the total area occupied by our benchmarks, with respect to the different memory allocation baselines. One tile size per benchmark is considered. Table 6.3 shows a breakdown of the area for the `jacobi-2d` benchmark with respect to layout/access pattern (naive, bounding box, MARS), and data type.

MARS introduces extra control logic and extra I/O functions that the other baselines do not have. It is therefore normal to observe area increases with this baseline. Figure 6.10a and Figure 6.10a show the LUT and FF utilization of the three considered benchmarks, varying the data type, for one tile size and one metadata size per benchmark. We observe that the MARS baseline consistently uses more LUT and FF than the bounding box baseline; however, this is not the case with respect to the original allocation. For instance, on the `seidel2d` benchmark, the original allocation baseline consumes 44% more LUT than the MARS baseline (34752 vs. 24000). This is explained by a complex control flow: tiles of that benchmark are skewed, resulting in a complex access pattern and control logic for the input. As Figure 6.8 shows, this baseline also has the lowest access performance due to the access pattern's complexity.

The logic area increases with the data type width. This is illustrated in Figure 6.10a. This increase is more sensible in `jacobi1d` where the on-chip arrays are implemented in logic instead of Block RAM, which effect is also visible in Figure 6.10d.

Regarding the DSP and BRAM utilization, Figure 6.10c and Figure 6.10d show that the MARS baseline consistently requires more DSP and BRAMs than the two other baselines.

Table 6.3: Occupied area for `jacobi-2d`, tile size 10x10x10. MARS versions contain a scratchpad memory for 11520 compression metadata.

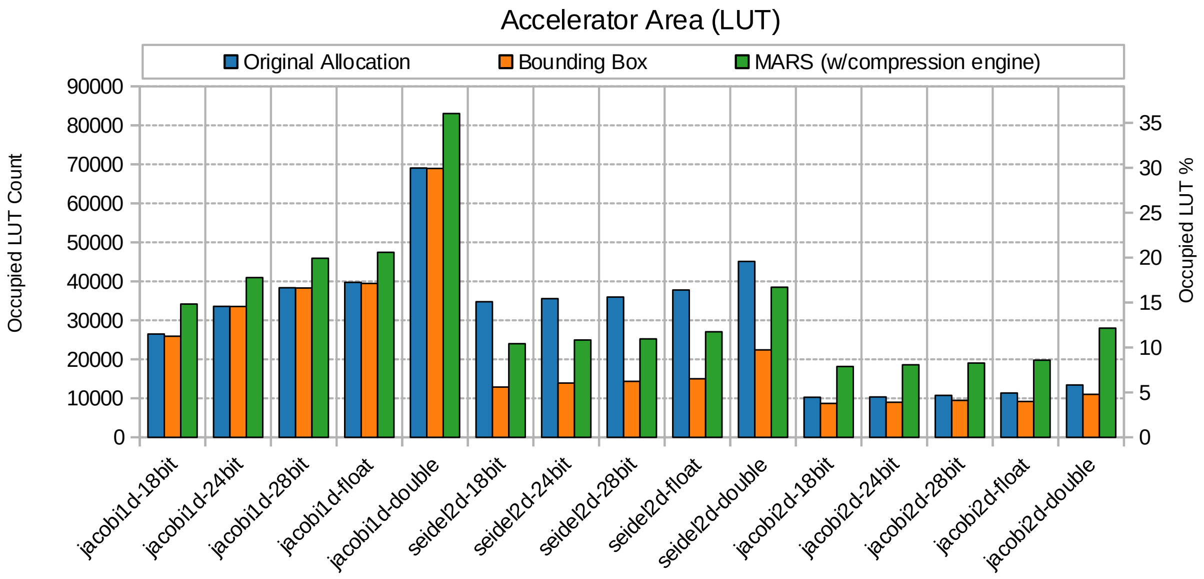
Layout	Data type	LUT	FF	BRAM	DSP
`naive`	18-bit	10265	11299	5	5
`naive`	24-bit	10345	11418	6	5
`naive`	28-bit	10746	11667	6	3
`naive`	`float`	11359	13073	6	14
`naive`	`double`	13411	15275	12	23
`B-Box`	18-bit	8693	10181	2	5
`B-Box`	24-bit	8980	10572	3	5
`B-Box`	28-bit	9471	11010	3	3
`B-Box`	`float`	9190	11246	3	10
`B-Box`	`double`	11012	13343	6	17
`MARS`	18-bit	18148	16437	87	42
`MARS`	24-bit	18589	16666	89	42
`MARS`	28-bit	19050	16877	89	40
`MARS`	`float`	19774	18342	89	51
`MARS`	`double`	27984	21508	102	60

This is expected: the MARS baseline performs all the I/O operations other baselines also perform, plus compression and on-chip data layout change. To this aim, FIFOs holding all the MARS data are implemented only on the MARS baseline, and require extra BRAMs. The extra DSP blocks for MARS baselines come from the address computations that are performed inside the I/O units. Indeed, the size of the space is passed as a parameter instead of being a constant, thus requiring true multipliers.

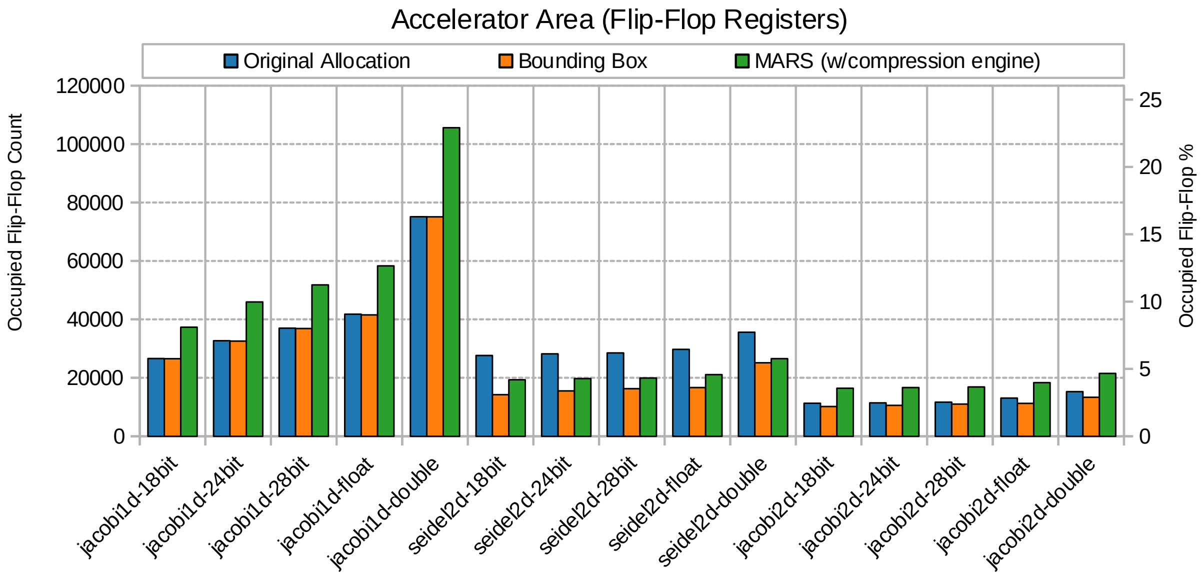
Applicability

Figure 6.11 shows the compression rate for each data type and tile size for the `jacobi1d` benchmark. Two ratios are shown: the *true ratio* which accounts only for the bit savings due to compression, and a *ratio with padding* that accounts for the savings due to not padding the data. The ratio with padding is the one our accelerators really benefit from, because the data is not packed in memory except in compressed MARS form.

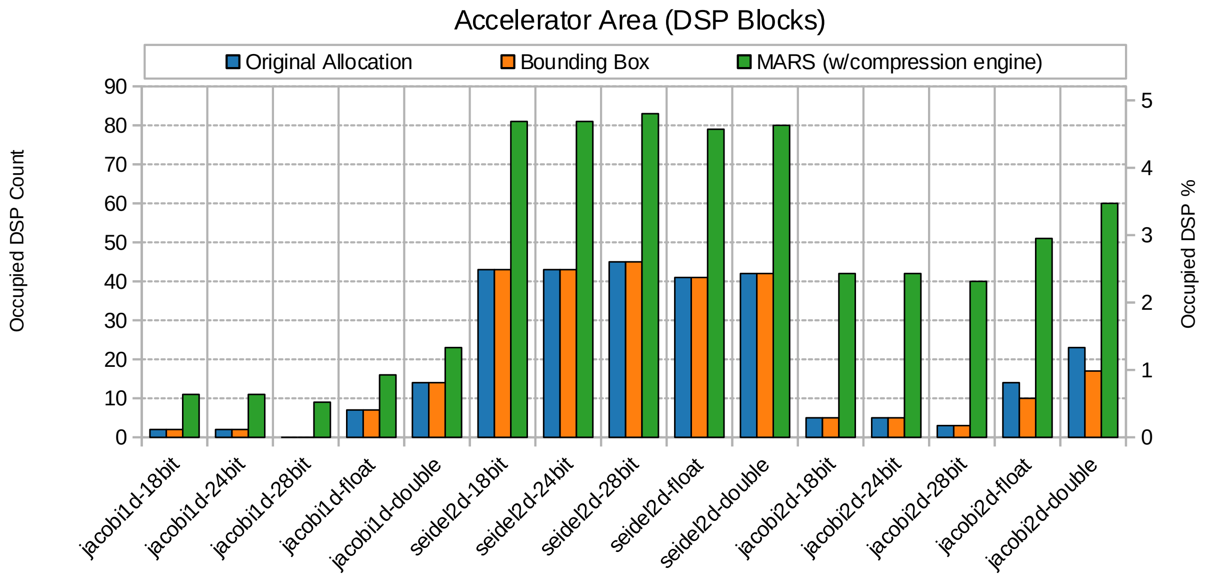
Overall, compressing the data for the selected benchmarks is almost always profitable, possibly largely as the compression ratio goes up to 5.09:1 for 200×200 tiles and 18-bit type.



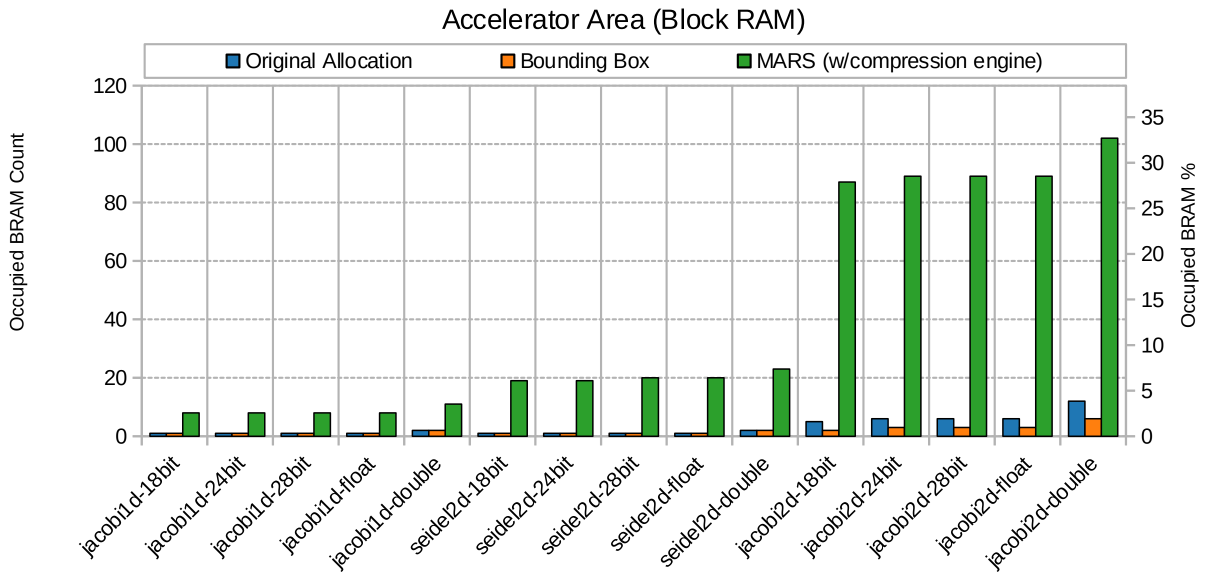
(a) Look-up tables



(b) Flip-flops



(c) DSP blocks



(d) Block RAM

Figure 6.10: Area statistics for the three benchmarks. Tile sizes are 200×200 for `jacobi1d`, $10 \times 10 \times 10$ for `jacobi2d` and $4 \times 10 \times 10$ for `seidel2d`.

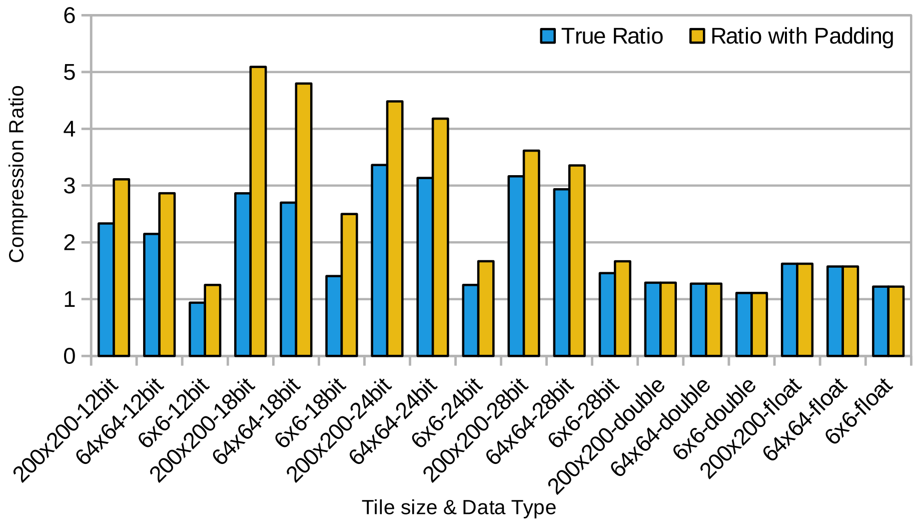


Figure 6.11: Compression ratio vs. data type and tile size for jacobi1d

We can observe that large tiles (64×64, 200×200) exhibit closer compression ratios than smaller tiles (6×6). This discrepancy can be explained by the compressed chunks being too small to benefit from the data's low entropy; for the smallest data type and tile size, compressing data is even worse than not compressing.

6.6 Conclusion

In this chapter, we introduce a novel global memory layout for FPGA accelerators that maximises contiguity of the accessed data under constraint of irredundancy. We evaluate our approach against access patterns derived from the original layout and show that a MARS-based partitonning of the data allow a substantial increase of throughput and decrease of transfer time at the cost of a minimal resource increase.

Chapter 7

An Irredundant Decomposition of Data Flow with Affine Dependences

7.1 Introduction

In the previous chapters, we have covered data transfer optimizations for polyhedral programs that have uniform dependences. The scope of these optimizations is limited to intermediate results produced by these programs, due to uniform dependences. In the polyhedral model, it is however possible to entirely determine the data movement of the program, and optimize it in two respects: first, reducing the amount of communication by exhibiting *locality*; second, by optimizing the existing communications to reduce their latency and better utilize the available bandwidth.

In this chapter, we broaden the scope of the analysis of Chapter 5, to also support input data. Like it is done for intermediate results, input data transfers need to be optimized for both locality and memory access performance. This is even more important when the same input data is reused and transferred multiple times to the accelerator over the course of an entire execution.

Dependences to input variables are rarely *uniform*, because the data arrays usually have less dimensions than the domain of computation. The existing dependence-based partitioning works must therefore be extended to support affine dependences to input variables, and to propose a re-allocation of these variables.

This chapter seeks to extend the partitioning of Chapter 5 to handle the entire data flow of the tile and maximize access contiguity. Its contributions are as follows:

- We propose a partitioning scheme, called Affine-MARS, of data spaces and iteration spaces with a pre-existing tiling,

- We formalize the construction of this partitioning scheme and determine its limitations.

This chapter is organized as follows: Section 7.2 justifies this work on partitioning iteration and data spaces; Section 7.3 gives the notions of MARS and the linear algebra concepts used throughout this work; Section 7.4 proposes construction methods for Affine-MARS according to the dependences; finally, Section 7.5 compares this approach to existing iteration- and data-space partitioning methods.

7.2 Motivation

The motivation of this work stems from two driving forces: the necessity to exhibit data access contiguity, and the limitations of existing analyses preventing efficient (coalesced) memory accesses.

7.2.1 Necessity of spatial locality

To motivate this work, we can consider a matrix multiplication program. At each step of its computations, it needs input values ($a_{i,k}$ and $b_{k,j}$), an intermediate result (partial sum of $c_{i,j}$) and produces a new result. Previous work has shown that using loop tiling increases the performance due to improved locality. When tiling is applied, the matrices are processed in "patches" as illustrated in Figure 7.1.

In this application, multiplying matrices $A = (a_{i,j})$ and $B = (b_{i,j})$ is done by computing all $a_{i,k} \times b_{k,j}$. Loop tiling, for locality, can be applied and gives a division of the space as in Figure 7.1.

In this example, an entire patch of each input matrix A, B is transferred for the execution of each tile.

Despite the added locality, the application can still be memory-bound: tiled matrix product lacks data access contiguity. Barring any data layout manipulations, data is contiguous within a row (for row-major storage) or column (for column-major storage). A patch of A, B or C is never contiguous because it contains *multiple parts* of contiguous rows (or columns).

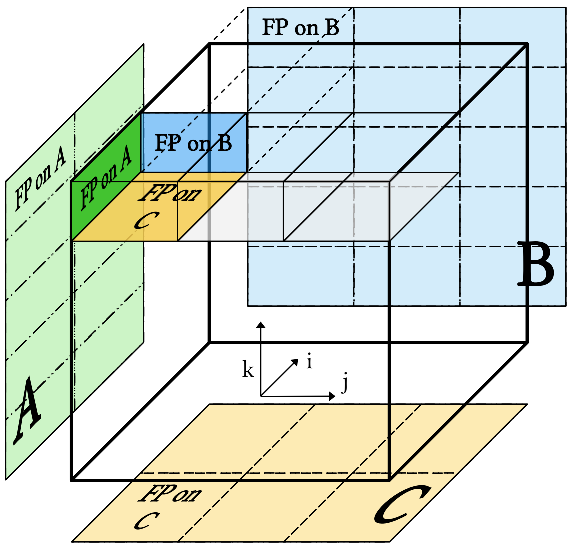


Figure 7.1: Tiled matrix product, and footprint of a tile on the matrices. Each footprint "patch" is composed of multiple contiguous rows or columns, but none of the patches are entirely contiguous in memory.

The lack of contiguity therefore induces multiple short burst accesses to retrieve the entire patch.

Like for intermediate results, it is desirable to increase spatial locality and leverage contiguity to obtain higher performance on the input variables. Data blocking has been known to increase the performance of matrix multiplication, especially because data block correspond exactly to the "footprint" of iteration tiles on the matrix.

7.2.2 Limitations of existing transformations

Although data tiling is sufficient for matrix multiplication, more complex computational patterns require a finer data partitioning.

Chapter 5 proposes a breakup of intermediate results of programs with purely uniform dependence patterns, that enables contiguity. However, such dependence patterns exclude commonly found affine dependences, such as the *broadcast*-type dependences of matrix multiplication, despite there existing a natural breakup like data blocking.

Moreover, automatic data blocking is mostly applied by domain-specific compilers that have to generate the memory allocation (e.g. Halide [62], AlphaZ [85]). When there exists a memory allocation and data layout in the input code, compilers follow it unless specific directives (e.g. the `ARRAY_PARTITION` directive in FPGA high-level synthesis tools based on [17]) are given to them. Allowing the compiler to change this allocation would open the door to better bandwidth utilization. Works on inter-node communication [23, 90] where memory allocation only exists within the nodes (and not across nodes) can resort to very specific groupings of data to minimize the communication overhead; it makes sense to apply this idea likewise to host-to-accelerator communications, despite there existing a global memory allocation.

In this work, we generalize the principle of data blocking to automatically partition the data arrays in function of when (in time) they are consumed. This approach can only be guaranteed to work with a specific class of programs called *polyhedral programs* where the exact data flow is known using static analysis.

In the same approach, we propose a secondary partitioning of the intermediate results; notably, this generalization coincides with an extension of the scope of Chapter 5 to affine dependences.

7.3 Background and hypotheses

We propose an automated approach to partitioning the data flow of a program. To construct it, we rely the polyhedral analysis and transformation framework and elements of linear algebra that this section introduces.

7.3.1 Polyhedral representation, tiling

To be eligible for affine MARS partitioning, a program (or a section thereof) must have a polyhedral representation. It may come either from the analysis of an imperative program (e.g. using PET [75] or Clan [4]) or a domain-specific language. In any case, the following elements from Table 3.1 are assumed to be available:

- A d-dimensional iteration domain $D \subset \mathbf{Z}^d$, or a collection of such domains,

- A k-dimensional data domain $\mathcal{A}$,
- A collection $(\varphi_i)_i$ of access functions $\varphi_i : D \rightarrow \mathcal{A}$, defining the reads and writes at each instance,
- A *polyhedral reduced dependence graph* (PRDG), constructed e.g. via array dataflow analysis [27].

The core elements extracted from the polyhedral representation are the dependences, that model which data must be available for a computation (any point in D) to take place. The data flow notably comprises two kinds of dependences we focus about in this chapter:

- Flow dependences: correspond to passing of intermediate results within the polyhedral section of the program,
- Input dependences: correspond to input data going into the program.

Both can be mapped to affine functions corresponding to the following definition:

Definition 1. *A* dependence function *is any affine function from an iteration domain D to another domain D' (iteration or data). In particular, a dependence function is a single-valued relation (each element of D has a single image).*

Each dependence will be noted B, and as an affine function, it is computed as $B(\vec{x}) = A\vec{x} + \vec{b}$ with A a matrix and $\vec{b}$ a vector.

Each *domain* is a subset of a *Euclidean vector space* $E \subset \mathbf{Z}^d$. In particular, every point $x \in D$ is associated to a vector $\vec{x} \in E$. Section 7.3.2 gives further elements of linear algebra used throughout this chapter.

In previous chapters, we were only relying on uniform dependences; in this chapter, we need a formal definition of an affine dependence, which is provided below.

Definition 2. *A dependence $B(\vec{x}) = A\vec{x} + \vec{b}$ is said to be* uniform *when A is the square identity matrix. A collection of dependences $B_1, \ldots, B_n$ are* uniformly intersecting *if they all have the same linear part, i.e. the same A matrix.*

To create a partitioning of the data spaces, our work relies on an existing partitioning of the iteration space. Loop tiling [41, 63, 68, 79], as introduced in Chapter 3, creates such a partitioning. It uses *tiling hyperplanes* to do so. Each hyperplane is defined by a *normal vector* (of unit norm). Tiles are periodically repeated, with a period s called the *tile size*. We notably use *scaled normal vectors* that translate a point from a tile to the same point in another tile by crossing one tiling hyperplane.

In this work, we assume **tiling hyperplanes are linearly independent from each other**. Each tile has (unique) coordinates that are represented by a t-dimensional vector $\vec{t} = (i_1, \ldots, i_t)$ where t is the number of tiling hyperplanes. This tile is the set defined by

$$T(\vec{t}) = \left\{ \vec{x} \in E : \bigwedge_{j \in \{1,\ldots,t\}} s_j i_j \leqslant \vec{x} \cdot \vec{n}_j < s_j(1 + i_j) \right\}$$

The *footprint* of a dependence B and a tile $T(\vec{t})$ is the image of the tile by the dependence:

$$B \left\langle T(\vec{t}) \right\rangle = \left\{ B(\vec{x}) : x \in T(\vec{t}) \right\}$$

7.3.2 Linear algebra

In this chapter, we use several fundamental results from linear algebra. Below are reminders of them for the reader's reference.

Spaces and bases

Definition 3. *Let E be an Euclidean vector space of d dimensions with its scalar product noted $\vec{x} \cdot \vec{y}$. Let $\mathcal{B} = (\vec{e}_1, \ldots, \vec{e}_d)$ be a basis of E. $\mathcal{B}$ is called an* orthonormal basis *of E when for all $i \neq j$, $\vec{e}_i \cdot \vec{e}_j = 0$ and for all i, $\vec{e}_i \cdot \vec{e}_i = 1$.*

Proposition 3. *Any Euclidean space E of d dimensions admits an orthonormal basis.*

The proof of Proposition 3 is done by applying the Gram-Schmidt basis orthonormalization to an existing basis.

Definition 4. *The vector space of all linear combinations of a number of vectors* $(\vec{e}_1, \ldots, \vec{e}_n)$ *is noted* $\mathsf{vect}(\vec{e}_1, \ldots, \vec{e}_n)$. *Notably, that space has up to* n *dimensions, and exactly* n *dimensions if all the* n *vectors are linearly independent.*

Definition 5. *Two subspaces* S_1 *and* S_2 *of a vector space* E *are* supplementary *into* E *when their intersection is the null vector* $\vec{0}$, *and there exists a decomposition of all* $\vec{x} \in E$ *as* $\vec{x}_1 + \vec{x}_2$ *with* $\vec{x}_1 \in S_1$ *and* $\vec{x}_2 \in S_2$. *That decomposition is notably unique.*

Linear applications

Definition 6. *Let* $A : E \to F$ *be a linear application. The subspace* K *of* E *such that* $\forall \vec{x} \in K, A\vec{x} = \vec{0}$ *is called the* null space *of* A *and is noted* $\mathsf{ker}(A)$.

Definition 7. *Let* $A : E \to F$ *be a linear application. The image of* E *by* A *is noted* $A\langle E\rangle$. *Likewise, the image of a subspace* $S \subset E$ *by* A *is noted* $A\langle G\rangle$. *The preimage of a subspace* $T \subset F$ *is noted* $A^{-1}\langle F\rangle$.

Proposition 4. *If* $A : E \to F$ *is a linear application,* E *has* d *dimensions, and* $\mathsf{ker}(A)$ *is its null space, then let* $k \leqslant d$ *be the dimensionality of* $\mathsf{ker}(A)$. *There exists a* $d - k$*-dimensional supplementary* I *of* $\mathsf{ker}(A)$ *in* E*, such that:*

$$\forall \vec{x} \in I, (A\vec{x} = \vec{0} \Rightarrow \vec{x} = \vec{0})$$

7.4 Partitioning Data and Iteration Spaces

This section constitutes the core of our work: it proposes a breakup of the iteration and data spaces based on the same properties as the existing uniform breakup, detailed in Section 7.4.1.

The reasoning leading to the MARS starts from a simple, restrictive case (one single dependence, Section 7.4.3) and progressively relaxes its hypotheses (multiple *uniformly intersecting* dependences, Section 7.4.4 and non-*uniformly intersecting* dependences, Section 7.4.5).

The last step of the reasoning in Section 7.4.6 adds the constraint of partitioning an existing tiled space, which allows to partition intermediate results.

7.4.1 Case of uniform dependences

Maximal Atomic irRedundant Sets (MARS) are introduced in Chapter 5. They are defined as a partition of the *flow-out* iterations of a tile, such that every element of the partition is the largest set of iterations that verifies:

- Atomicity: consumption of a single element from a MARS implies consumption of the entire MARS.
- Maximality: considering all the consumers of a MARS M_0 (C_0) and all the consumers of another MARS M_1 (C_1), if $C_0 = C_1$, then $M_0 = M_1$.
- Irredundancy: each element of the MARS space belongs to no more than a single MARS.

While Chapter 5 uses the flow-in and flow-out information in the sense of [7], input data and output data do not belong to this information. This work instead resorts on the notion of *footprint* from [2]; notably, the notion of *flow-in iterations* of a tile coincides with that of a footprint of a tile (of iterations) on another tile of iterations.

The properties of MARS constructed with uniform dependences are the same as those sought in this chapter. Merely proposing a partition of the iteration or data spaces satisfies the irredundancy property; the properties to actually check from the partitioning are the atomicity and maximality.

7.4.2 The problem: uniform versus affine dependences

In the uniform case, MARS can be constructed by enumerating all the *consumer tiles* of a given tile, i.e. those other tiles that need data from that tile. The uniformity guarantees that there are a finite number of consumer tiles, and that all tiles will exhibit the same MARS regardless of their position in the iteration space (i.e. MARS are invariant by translation of a tile).

Affine dependences do not guarantee a finite number of consumer tiles; it may be parametric or potentially unbounded. Also, it becomes necessary to assert when the invariance by translation is possible.

In the rest of this section, we will prove, for one and multiple dependences:

- The **existence of a finite set of representatives of all consumer tiles**, suitable to determine the MARS partition,
- The **invariance** of the partitioning **by a translation of a tile**, or conditions to guarantee it.

7.4.3 Case of a single affine dependence

The simplest case is when there is a single affine dependence between a tiled iteration space and a data space. This subsection starts with an example and explains the general case afterwards.

Example

To start with, we introduce an example with a single dependence, and non-canonical tiling hyperplanes.

- Domain: $\{(i, j) : 0 \leqslant i < N, 0 \leqslant j < M\}$, basis vectors $\vec{e_i}, \vec{e_j}$
- Dependence : $S_0(i, j) \mapsto \mathcal{A}(i)$, represented as $B(i, j) = (i)$ (i.e. $B(\vec{x}) = A\vec{x} + \vec{b}$ with $A : (i, j) \mapsto (i)$ and $\vec{b} = \vec{0}$).
- Tiling hyperplanes : $H_0 : i + j$, $H_1 : j - i$
- Normal vectors: $\vec{n}_1 = (1, 1)$, $\vec{n}_2 = (-1, 1)$; scaled normal vectors (w.r.t. tile size): $\vec{\mathbf{n}}_1 = (s/2, s/2)$, $\vec{\mathbf{n}}_2 = (-s/2, s/2)$

- Tile size : $s \in \mathbf{N}^*$

We want to construct the MARS on the $\mathcal{A}$ data space. To do so, we are going to compute the *footprint* [2] of a tile onto the data space along the dependence B; then, by noticing that all footprints are a translation of the same footprint, we will determine parametrically which tiles have intersecting footprints, and compute the MARS using the same method as Chapter 5.

We first define a tile of iterations with a parametric set : we call $T(i_0, i_1)$ the set :

$$T(i_0, i_1) = \{(i, j) : si_0 \leqslant i + j < s(1 + i_0) \wedge si_1 \leqslant j - i < s(1 + i_1)\}$$

The footprint of $T(i_0, i_1)$ by the dependence B, appearing in Figure 7.2, is therefore:

$$B \langle T(i_0, i_1) \rangle = \{(i) : \exists j : si_0 \leqslant i + j < s(1 + i_0) \wedge si_1 \leqslant j - i < s(1 + i_1)\}$$

where the existential quantifier may be removed:

$$B \langle T(i_0, i_1) \rangle = \{(i) : s(i_0 - i_1 - 1) < 2i < s(i_0 - i_1 + 1)\}$$

Given (i_0, i_1), we now seek the other tiles which footprint's intersection with $B \langle T(i_0, i_1) \rangle$ is not empty: let (i_2, i_3) be another tile.

$$\begin{aligned}
&B \langle T(i_0, i_1) \rangle \cap B \langle T(i_2, i_3) \rangle = \\
&\{(i) : s(i_0 - i_1 - 1) + 1 \leqslant 2i \leqslant s(i_0 - i_1 + 1) - 1 \\
&\wedge s(i_2 - i_3 - 1) + 1 \leqslant 2i \leqslant s(i_2 - i_3 + 1) - 1\}
\end{aligned}$$

The intervals $[\![s(i_0 - i_1 - 1) + 1; s(i_0 - i_1 + 1) - 1]\!]$ and $[\![s(i_2 - i_3 - 1) + 1; s(i_2 - i_3 + 1) - 1]\!]$ intersect if $i_0 - i_1 = i_2 - i_3 + 1$, $i_0 - i_1 = i_2 - i_3$ or $i_0 - i_1 = i_2 - i_3 - 1$.

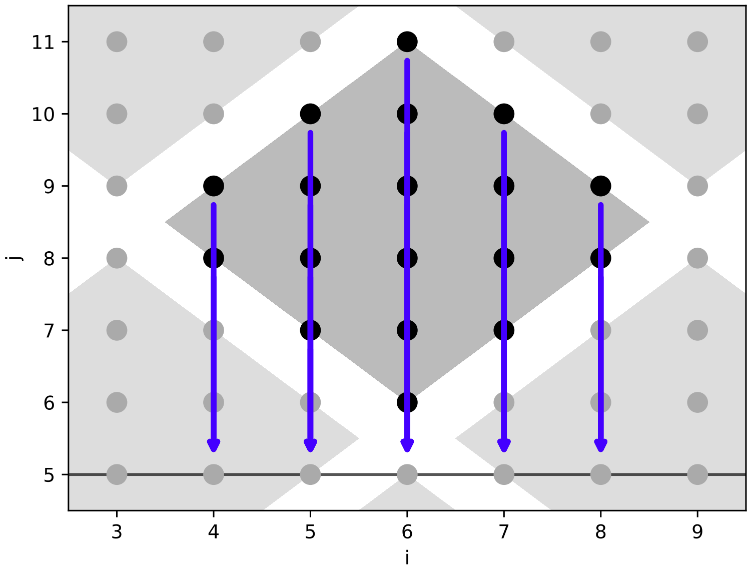


Figure 7.2: Footprint of one tile with a single affine dependence $B(i, j) = (i)$. The one-dimensional destination space is shown as a continuous line.

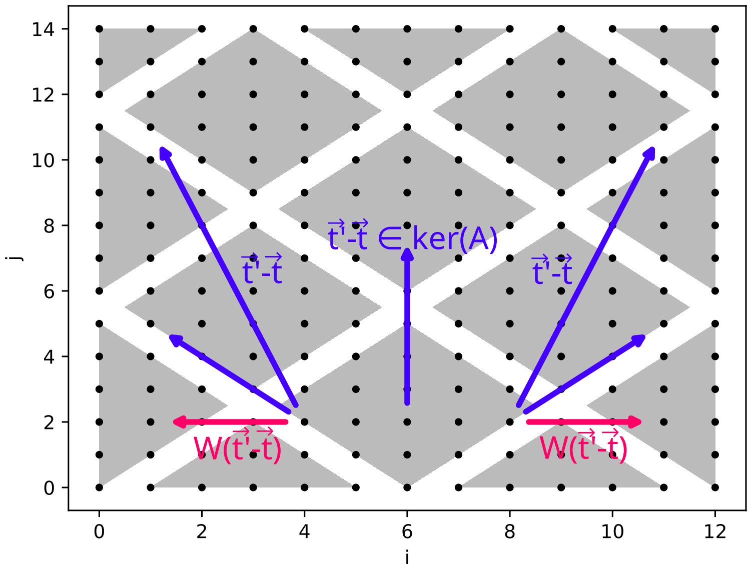


Figure 7.3: Some consumer tiles of one tile $T(\vec{t})$ with a single dependence $B(i, j) = (i)$, and the projection $W(\vec{t'} - \vec{t})$ on a supplementary of $\mathsf{ker}(A)$. There are a finite number of such projected vectors, and they are constant.

The valid (i_2, i_3)s are therefore:

$$\begin{aligned}(i_2, i_3) \in \{(i_0 + p, i_1 + p); p \in \mathbf{Z}\} \\ \cup \{(i_0 + p - 1, i_1 + p); p \in \mathbf{Z}\} \\ \cup \{(i_0 + p + 1, i_1 + p); p \in \mathbf{Z}\}\end{aligned}$$

as shown in blue in Figure 7.3.

The space of valid (i_2, i_3) is infinite: we can visually see this as all tiles along a vertical axis share the same footprint on $\mathcal{A}$. We can formalize this intuition by computing the kernel of A : in this case, it is

$$\mathsf{ker}(A) = \mathsf{vect}(\vec{e}_j)$$

and the image of a point on $\mathcal{A}$ is invariant by any upwards or downwards translation.

There are however only three *distinct* footprints intersecting with that of $T(i_0, i_1)$; the other footprints stem from tiles which are translations along $\mathsf{ker}(A)$. These footprints come from the top-left, top-right tiles and all tiles above them vertically; these consumer tiles are shown in Figure 7.3.

We can decompose the space (i, j) using a basis of the kernel and a supplementary, for instance $E = \mathsf{vect}(\vec{e}_i) \oplus \mathsf{ker}(A)$.

In this basis, we can express the coordinates of tile's origins for the case where $(i_2, i_3) = (i_0 + p + 1, i_1 + p)$ with $p \in \mathbf{Z}$ using the scaled normal vectors $\vec{\mathbf{n}}_1$, $\vec{\mathbf{n}}_2$:

$$\begin{aligned}i_2\vec{\mathbf{n}}_1 + i_3\vec{\mathbf{n}}_2 &= i_2\frac{s}{2}(\vec{e}_i + \vec{e}_j) + i_3\frac{s}{2}(\vec{e}_j - \vec{e}_i) \\ &= \frac{s}{2}(i_0 + p + 1)(\vec{e}_i + \vec{e}_j) + \frac{s}{2}(i_1 + p)(\vec{e}_j - \vec{e}_i) \\ &= \frac{s}{2}(i_0 - i_1 + 1)\vec{e}_i + \frac{s}{2}(i_0 + i_1 + 2p + 1)\vec{e}_j\end{aligned}$$

which, when projected onto $\mathsf{vect}(\vec{e}_i)$, gives:

$$P_{\mathsf{vect}(\vec{e}_i)}(i_2\vec{\mathbf{n}}_1 + i_3\vec{\mathbf{n}}_2) = \frac{s}{2}(i_0 - i_1 + 1)\vec{e}_i$$

which is independent of p. This means that all points within tile $T(i_2, i_3)$ have the same image by B. Therefore, **given** (i_0, i_1)**, the entire family of tiles** $(i_0 + p + 1, i_1 + p)$ **have the same footprint on $\mathcal{A}$.** We can therefore consider a single representative of that family to compute the MARS.

Likewise, if $(i_2, i_3) = (i_0 + p - 1, i_1 + p)$ with $p \in \mathbf{Z}$, then

$$P_{\mathsf{vect}(\vec{e}_i)}(i_2\vec{\mathbf{n}}_1 + i_3\vec{\mathbf{n}}_2) = \frac{s}{2}(i_0 - i_1 - 1)\vec{e}_i$$

which is also independent of p; the same conclusion holds for $(i_2, i_3) = (i_0 + p, i_1 + p)$ and $P_{\mathsf{vect}(\vec{e}_i)}(i_2\vec{\mathbf{n}}_1 + i_3\vec{\mathbf{n}}_2) = \vec{0}$. Figure 7.3 shows in pink the projection of the *translation vectors* from the tile $T(\vec{t})$ to its consumers (there are only two non-null projections, so only two such vectors appear).

There are an infinity of tiles which footprint intersects with that of a given tile; however, to compute the MARS, we have demonstrated that it is sufficient to take three representative tiles. The same procedure as in Chapter 5 can be applied once these three *consumer tiles* have been determined.

Per Algorithm 3, we compute the respective intersections with $B\langle T(i_0, i_1)\rangle$ of all other consumer tiles: for $(i_2, i_3) = (i_0 + p + 1, i_1 + p)$ with $p \in \mathbf{Z}$,

$$\begin{aligned} &B\langle T(i_0, i_1)\rangle \cap B\langle T(i_0 + p + 1, i_1 + p)\rangle = \\ &\{(i) : s(i_0 - i_1 - 1) < 2i < s(i_0 - i_1 + 1) \\ &\qquad \wedge s(i_0 - i_1) < 2i < s(i_0 - i_1 + 2)\} \\ &= \{(i) : s(i_0 - i_1) < 2i < s(i_0 - i_1 + 1)\} \end{aligned}$$

When $(i_2, i_3) = (i_0 + p - 1, i_1 + p)$ with $p \in \mathbf{Z}$,

$$\begin{aligned} &B\langle T(i_0, i_1)\rangle \cap B\langle T(i_0 + p - 1, i_1 + p)\rangle = \\ &= \{(i) : s(i_0 - i_1 - 1) < 2i < s(i_0 - i_1)\} \end{aligned}$$

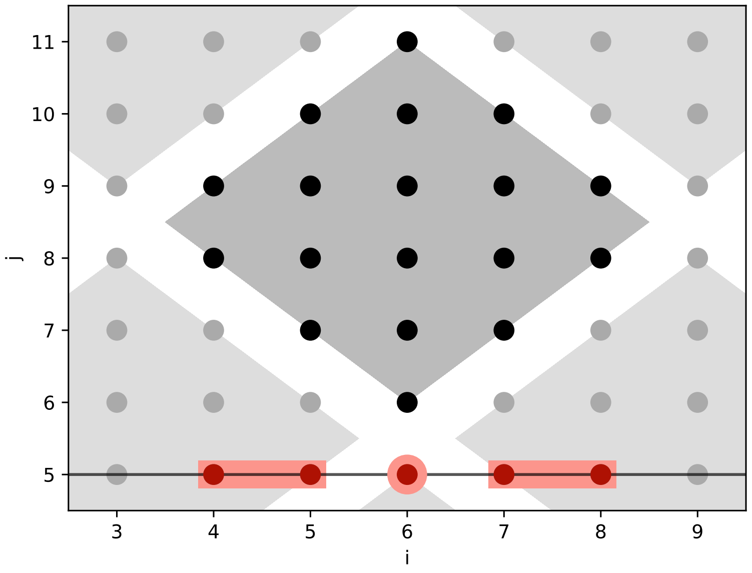


Figure 7.4: MARS obtained with a single affine dependence $B(i, j) = (i)$.

Finally, when $(i_2, i_3) = (i_0 + p, i_1 + p)$,

$$B \langle T(i_0, i_1) \rangle \cap B \langle T(i_0 + p, i_1 + p) \rangle =$$

$$= \{(i) : s(i_0 - i_1) = 2i\}$$

Also, $B \langle T(i_0 + p + 1, i_1 + p) \rangle \cap B \langle T(i_0 + p - 1, i_1 + p) \rangle = \varnothing$, so we have all the MARS.

The MARS on symbol $\mathcal{A}$ for this program seen from a tile $T(i_0, i_1)$ are therefore the three sets $\{(i) : s(i_0 - i_1) < 2i < s(i_0 - i_1 + 1)\}$, $\{(i) : s(i_0 - i_1 - 1) < 2i < s(i_0 - i_1)\}$ and $\{(i) : s(i_0 - i_1) = 2i\}$. These MARS are shown in Figure 7.4.

General case

In the general case, computing the MARS for a single dependence leading to a non-tiled space can be done as follows. Let D be the d-dimensional iteration space from which the dependence originates, and E be the vector space such that $D \subset E$.; let $\mathcal{A}$ be the destination space. Let B be the dependence with $B(\vec{x}) = A\vec{x} + \vec{b}$. Let $(H_1, \ldots, H_t)$ be the t

tiling hyperplanes, $(\vec{n}_1, \ldots, \vec{n}_t)$ normal vectors to the tiling hyperplanes, $(s_1, \ldots, s_t)$ be the tile sizes.

For a tile coordinate be $\vec{t} = (i_1, \ldots, i_t)$, the tile is defined as

$$T(\vec{t}) = \{\vec{x} = (x_1, \ldots, x_d) : \forall j \in \{1, \ldots, t\} : s_j i_j \leqslant \vec{x} \cdot \vec{n}_j < s_j(1 + i_j)\}$$

We can compute, with $\vec{t}$ and another tile $\vec{t'}$ as a parameter, when the intersection of $B\left\langle T(\vec{t})\right\rangle$ and $B\left\langle T(\vec{t'})\right\rangle$ is non-empty using affine operations. Let $V(\vec{t})$ be:

$$V(\vec{t}) = \left\{\vec{t'} : B\left\langle T(\vec{t})\right\rangle \cap B\left\langle T(\vec{t'})\right\rangle \neq \varnothing\right\}$$

which is obtainable by taking the parameter space of $I(\vec{t}, \vec{t'})$. $V(\vec{t})$ represents the tile coordinates of all tiles which footprint on $\mathcal{A}$ intersects that of $T(\vec{t})$.

In Chapter 5, $V(\vec{t})$ is determined by browsing through neighboring tiles. The main difficulty here is that $V(\vec{t})$ **is potentially infinite.** We will demonstrate that there are only a finite number of *distinct footprints* overlapping with $B\left\langle T(\vec{t'})\right\rangle$. To determine them, we suggest to decompose E into $\mathsf{ker}(A)$ and a supplementary I of $\mathsf{ker}(A)$, i.e.

$$E = I \oplus \mathsf{ker}(A)$$

Proposition 4 gives us that such a decomposition always exists, and per Proposition 3, there is an orthonormal basis of the resulting space.

If $r = \mathsf{rank}(A)$, let $(\vec{e}_1, \ldots, \vec{e}_r)$ be a basis of I and $(\vec{e}_{r+1}, \ldots, \vec{e}_d)$ be a basis of $\mathsf{ker}(A)$ such that $(\vec{e}_1, \ldots, \vec{e}_d)$ is an orthonormal basis of E. Let $(\vec{n}_1^p, \ldots, \vec{n}_t^p)$ be the orthogonal projections of the $\vec{n}_i$s onto $(\vec{e}_1, \ldots, \vec{e}_r)$; in particular, these have zero $r+1$-th through d-th coordinates.

For any $\vec{t'} \in V(\vec{t})$, if $\vec{t'} - \vec{t} = (\delta_1, \ldots, \delta_t)$, we can compute

$$W(\vec{t'} - \vec{t}) = \sum_{i=1}^{t} \delta_i \vec{n}_i^p$$

which represents the part of the translation between tiles that results in translating the images.

The most important result needed to construct the MARS is the ability to enumerate *all* the footprints. The following proposition formalizes it:

Proposition 5. *The set* $P(\vec{t}) = \left\{W(\vec{t'} - \vec{t}) : \vec{t'} \in V(\vec{t})\right\}$ *is finite, and for each consumer tile* $\vec{t'} \in V(\vec{t})$*, there exists a unique* $\vec{p} \in P(\vec{t})$ *such that*

$$B\left\langle T(\vec{t'})\right\rangle = \left\{\vec{y} + A\vec{p} : \vec{y} \in B\left\langle T(\vec{t})\right\rangle\right\}$$

and that $\vec{p}$ *is a constant vector, independent of* $\vec{t}$ *(i.e. the consumer tiles are invariant by translation).*

Proof. **Completeness of footprints:** Let $\vec{t'} \in V(\vec{t})$, i.e. a tile which footprint intersects that of tile $\vec{t}$. We know that $W(\vec{t'} - \vec{t}) = \sum_{i=1}^{t}(t'_i - t_i)\vec{n}_i^p = \vec{p} \in P(\vec{t})$. Then:

$$\begin{aligned}
B\left\langle T(\vec{t'})\right\rangle &= \left\{A\vec{x} + \vec{b} : \forall i : s_i t'_i \leqslant \vec{x}\cdot\vec{n}_i \leqslant (1 + t'_i)s_i\right\} \\
&= \left\{A\vec{x} + \vec{b} : \forall i : s_i(t_i + (t'_i - t_i)) \leqslant \vec{x}\cdot\vec{n}_i \leqslant (1 + t_i + (t'_i - t_i))s_i\right\} \\
&= \left\{A\left(\vec{x} + \textstyle\sum_{i=1}^{t}(t'_i - t_i)\vec{n}_i\right) + \vec{b} : \forall i : s_i t_i \leqslant \vec{x}\cdot\vec{n}_i \leqslant (1 + t_i)s_i\right\} \\
&= \left\{A\left(\vec{x} + \textstyle\sum_{i=1}^{t}(t'_i - t_i)\vec{n}_i^p\right) + \vec{b} : \vec{x} \in T(\vec{t})\right\} \\
&= \left\{\vec{y} + A\vec{p} : \vec{y} \in B\left\langle T(\vec{t})\right\rangle\right\}
\end{aligned}$$

using the fact that $A(\vec{n}_i) = A(\vec{n}_i^p)$.

Uniqueness of $\vec{p}$**:** A is bijective between I (supplementary of $\mathsf{ker}(A)$ in E) and $\mathsf{Im}(A)$. Therefore, because $\vec{p} \in P(\vec{t})$ is in I, it is the unique element of I which $A\vec{p}$ is the image. Therefore, $\vec{p}$ is unique in the sense of this proposition.

Finiteness of $P(\vec{t})$: For all $\vec{t'} \in V(\vec{t})$, $B\left\langle T(\vec{t'})\right\rangle$ is a translation of $B\left\langle T(\vec{t})\right\rangle$ by $A\vec{p}$ with some $\vec{p} \in P(\vec{t})$. The coordinates of $\vec{t}$ are integers, therefore $A\vec{p}$ is an integer linear combination of the $A\vec{n}_i^p$s for $i \in \{1, \ldots, t\}$. $T(\vec{t})$ being bounded, the footprint $B\left\langle T(\vec{t})\right\rangle$ is bounded, therefore only a finite number of translations of itself by $A\vec{p}$s intersect with it.

Constantness of $\vec{p}$: Let $\vec{t_0}, \vec{t_1} \in \mathbf{Z}^t$ and $\vec{t'}_0 \in V(\vec{t_0})$, and $\vec{p} = W(\vec{t'} - \vec{t})$. Let $\vec{t'}_1 = \vec{t_1} + (\vec{t'}_0 - \vec{t_0})$. Then:

$$\begin{aligned}
B\left\langle T(\vec{t'}_1)\right\rangle &= \left\{A\left(\vec{x} + \textstyle\sum_{i=1}^{t}(t'_{1i} - t_{1i})\vec{n}_i\right) + \vec{b} : \forall i : s_i t_{1i} \leqslant \vec{x}\cdot\vec{n}_i \leqslant (1 + t_{1i})s_i\right\} \\
&= \left\{A\left(\vec{x} + \textstyle\sum_{i=1}^{t}(t_{1i} - t_{1i} + (t'_{0i} - t_{0i}))\vec{n}_i\right) + \vec{b} : \vec{x} \in T(\vec{t_1})\right\} \\
&= \left\{A\vec{x} + \textstyle\sum_{i=1}^{t}(t'_{0i} - t_{0i})A\vec{n}_i + \vec{b} : \vec{x} \in T(\vec{t_1})\right\} \\
&= \left\{B(\vec{x}) + A\vec{p} : \vec{x} \in T(\vec{t_1})\right\}
\end{aligned}$$

which means that the translation between the images of $T(\vec{t_1})$ and $T(\vec{t'}_1)$ is the same as that of $T(\vec{t_0})$ and $T(\vec{t'}_0)$.

□

We can therefore enumerate $P(\vec{t})$, knowing that for each $\vec{w} \in P(\vec{t})$, $P^{-1}(\vec{w})$ represents *consumer tiles* that all have the same footprint by B. That footprint is computed as follows:

$$\Phi(\vec{w}) = B\left\langle T(\vec{t}) + \vec{w}\right\rangle \text{ where } T(\vec{t}) + \vec{w} = \left\{\vec{x} + \vec{w} : \vec{x} \in T(\vec{t})\right\}$$

We can then compute the MARS. For all the combinations of $\vec{w}$s, i.e. for all $C \in \mathcal{P}\left(P(\vec{t})\right)$, we determine the MARS associated with that combination of consumer tiles:

$$\mathcal{M}_C = \bigcap_{\vec{w}\in C}\left(\Phi(\vec{w}) \cap B\left\langle T(\vec{t})\right\rangle\right) \setminus \bigcup_{\vec{w}\notin C}\left(\Phi(\vec{w}) \cap B\left\langle T(\vec{t})\right\rangle\right)$$

7.4.4 Case of multiple, uniformly intersecting dependences

The previous section studied the case of a single dependence leading to a data space. We look at the more frequent case of multiple dependences that all have the same linear part.

General case

If the dependences are uniformly intersecting, they all have the same linear part. This means that they all have the same null space, and therefore the space decomposition into $E = I \oplus \mathsf{ker}(A)$ still applies.

Let there be Q dependences $B_1, \ldots, B_Q$ that are uniformly intersecting. This means that there exists an unique matrix A such that:

$$\forall i : B_i(\vec{x}) = A\vec{x} + \vec{b}_i$$

Because all tiles share the same linear part, the space of consumer tiles for each dependence will be the same *up to a translation.* Their linear part will notably be the same, and the same argument as in the case above holds to guarantee that $P(\vec{t}) = \left\{W(\vec{t'} - \vec{t}) : \vec{t'} \in V(\vec{t})\right\}$ is finite.

Let, by abuse of the notation, $B\left\langle T(\vec{t})\right\rangle$ be the combined footprint of all dependences:

$$B\left\langle T(\vec{t})\right\rangle = \bigcup_{i=1}^{Q} B_i\left\langle T(\vec{t})\right\rangle$$

For each dependence B_i with $i \in \{1, \ldots, Q\}$, we therefore compute $V_i(\vec{t})$ by intersecting $B_i\left\langle T(\vec{t'})\right\rangle$ and $B\left\langle T(\vec{t})\right\rangle$ (i.e. we want the intersection of the footprint of *one dependence* and the footprint of *all other dependences*); let

$$V(\vec{t}) = \bigcup_{i=1}^{Q} V_i(\vec{t})$$

The same decomposition $E = \mathsf{vect}(\vec{e}_1, \ldots, \vec{e}_r) \oplus \mathsf{vect}(\vec{e}_r + 1, \ldots, \vec{e}_d)$ is applicable due to all B_is sharing the same linear part A.

We can give a more meaningful expression for $P(\vec{t})$:

$$P(\vec{t}) = \left\{ W(\vec{t'} - \vec{t}) : \vec{t'} \in \bigcup_{i=1}^{Q} V_i(\vec{t}) \right\}$$

which means that $P(\vec{t})$ is composed of the projections of the vectors leading to any consumer tile of *any dependence* (and therefore takes into account the uniform translations between dependences).

The MARS can be computed by using $P(\vec{t})$. There are two differences with the case when there is only a single dependence:

- The footprints of the consumer tiles $\Phi(\vec{w})$ are specific to each dependence,
- The footprint of the tile $\vec{t}$ is the *union* of the footprint of all dependences.

For $i \in \{1, \ldots, Q\}$, let $\Phi_i(\vec{w})$ be:

$$\Phi_i(\vec{w}) = B_i \left\langle T(\vec{t}) + \vec{w} \right\rangle \text{ where } T(\vec{t}) + \vec{w} = \left\{ \vec{x} + \vec{w} : \vec{x} \in T(\vec{t}) \right\}$$

The MARS are constructed by taking all subsets of consumer tiles from $P(\vec{t})$, and looking at the points consumed *only* by these tiles.

Formally, let the cardinality of $P(\vec{t})$ be $\#C$. For all $K : 1 \leqslant K \leqslant \#C$ and all permutations σ of $\{1, \ldots, \#C\}$, let

$$C = \left\{ \vec{t}_{\sigma(1)}, \ldots, \vec{t}_{\sigma(K)} \right\} \text{ and } \overline{C} = \{ \vec{t}_{\sigma(K+1)}, \ldots, \vec{t}_{\sigma(\#C)} \}$$

Then, a MARS is constructed according to the following rules:

- For each consumer tile coordinates $\vec{t'} \in C$, there exists a dependence leading to $T(\vec{t'})$,
- No dependence leads to a consumer tile $\vec{t'} \in \overline{C}$

These two conditions to form a MARS can be written as:

$$\mathcal{M}_C = \bigcap_{\vec{w} \in C} \left(\bigcup_{i=1}^{Q} \Phi_i(\vec{w}) \cap B \left\langle T(\vec{t}) \right\rangle \right) \setminus \bigcup_{\vec{w} \in \overline{C}} \left(\bigcup_{i=1}^{Q} \Phi_i(\vec{w}) \cap B \left\langle T(\vec{t}) \right\rangle \right)$$

and there are at most $\mathsf{card}(\mathcal{P}(P(\vec{t}))) = 2^{\mathsf{card}(P(\vec{t}))}$ Cs and therefore as many MARS.

Example: uniform dependences

In this paragraph, we show that the computation of MARS using Chapter 5 coincides with that proposed in this chapter when the dependences are uniform. Such dependences are a special case of uniformly intersecting dependences, with a linear part being identity. Note that the destination space is considered to be a data space, and therefore dependences within a tile are counted in the footprint (self-consumption of data produced by a tile is dealt with in the next section).

Consider the Jacobi 1D example:

- Domain: $\{(i,j) : 0 \leqslant i < N, 0 \leqslant j < M\}$, basis vectors $\vec{e_i}, \vec{e_j}$
- Dependences : $B_1(i,j) = (i-1, j-1)$, $B_2(i,j) = (i, j-1)$, $B_3(i,j) = (i+1, j-1)$
- Tiling hyperplanes : $H_1 : i+j$ $(\vec{n}_2 = (1,1))$, $H_2 : j-i$ $(\vec{n}_2 = (-1,1))$
- Tile size : $s \in \mathbf{N}^*$

We compute the unified footprint $B \left\langle T(\vec{t}) \right\rangle$:

$$\begin{aligned} B \left\langle T(\vec{t}) \right\rangle = \{(i,j) : & \; s i_1 \leqslant i + j + (2-p) < s(1+i_1) \\ & \wedge s i_2 \leqslant j - i + p < s(1+i_2) : p \in \{0,1,2\}\} \end{aligned}$$

Notably, if we confuse the data space $\mathcal{A}(i,j)$ and the iteration space (i,j) (that is, each cell of $\mathcal{A}$ contains the result of one iteration), and we restrict the footprint to those points outside tile $T(\vec{t})$, we obtain the *flow-in* of that tile as in Figure 7.5, corresponding to the same definition as in Chapter 5.

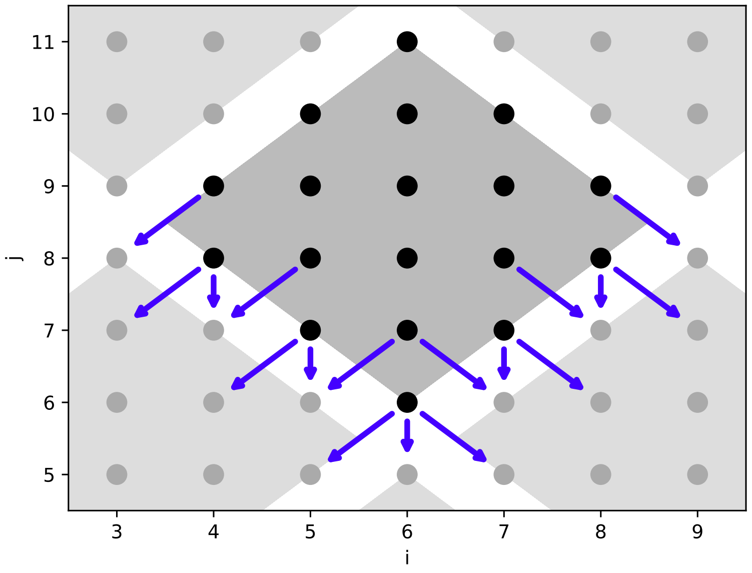


Figure 7.5: Flow-in dependences of tile $T(\vec{t})$ with uniformly intersecting dependences (Jacobi 1D).

We determine the individual $V_i(\vec{t})$s:

$$V_1(\vec{t}) = \{(i_1, i_2 - 1), (i_1, i_2), (i_1 + 1, i_2 - 1), (i_1 + 1, i_2)\}$$

$$V_2(\vec{t}) = \{(i_1, i_2), (i_1 + 1, i_2), (i_1, i_2 - 1), (i_1 - 1, i_2), (i_1, i_2 + 1)\}$$

$$V_3(\vec{t}) = \{(i_1, i_2), (i_1 - 1, i_2), (i_1, i_2 + 1), (i_1 - 1, i_2 + 1)\}$$

which gives

$$\begin{aligned} V(\vec{t}) = \{&(i_1, i_2), (i_1 - 1, i_2), (i_1, i_2 + 1), (i_1 - 1, i_2 + 1), (i_1 + 1, i_2), \\ &(i_1, i_2 - 1), (i_1 + 1, i_2 - 1)\} \end{aligned}$$

As $\mathsf{ker}(A) = \{0\}$, we easily get that $E = \mathsf{vect}(\vec{e}_i, \vec{e}_j)$ and therefore constructing the $W(\vec{t'} - \vec{t})$ is straightforward, yielding the following $P(\vec{t})$:

$$P(\vec{t}) = \{(0, 0), (-1, 0), (0, 1), (-1, 1), (1, 0), (0, -1), (1, -1)\}$$

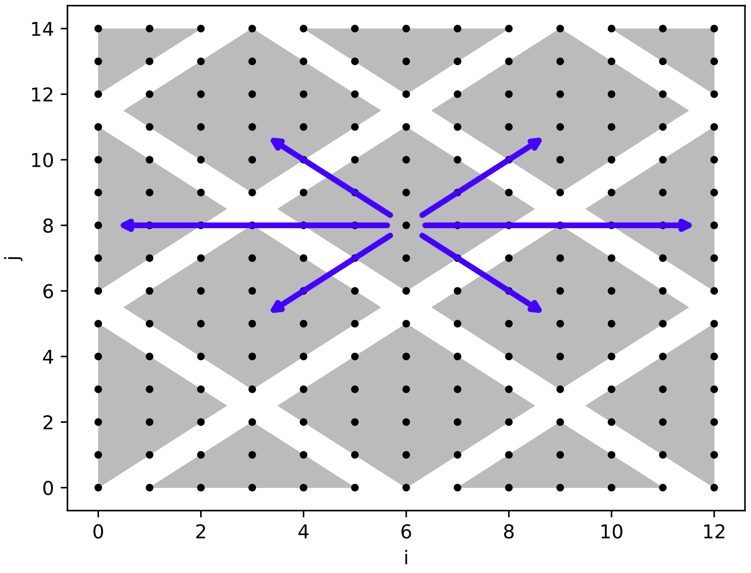


Figure 7.6: Consumer tiles of array $\mathcal{A}$ with Jacobi 1D dependences, sharing their footprint with tile $T(\vec{t})$.

This $P(\vec{t})$ means there are seven tiles (including $\vec{t}$ itself) which footprint (i.e. *any dependence*) intersects with $B\left\langle T(\vec{t})\right\rangle$. These consumer tiles are shown in Figure 7.6.

For the sake of shortness, we will not enumerate all combinations of consumer tiles. The MARS that appear after partitioning the footprints stemming from all consumer tiles are shown in Fig.7.7.

Again, if we confuse the iteration and data spaces, we can obtain the same MARS as computed in Chapter 5 by removing those MARS that are contained within $T(\vec{t})$; the result of this operation, shown in Figure 7.8, illustrates the coincidence between the MARS computed using uniform and affine dependences.

7.4.5 Case of multiple, non-uniformly intersecting dependences

We now consider the case where the dependences are not uniformly intersecting. In this case, the main difference is that dependences no longer share the same linear part. Therefore,

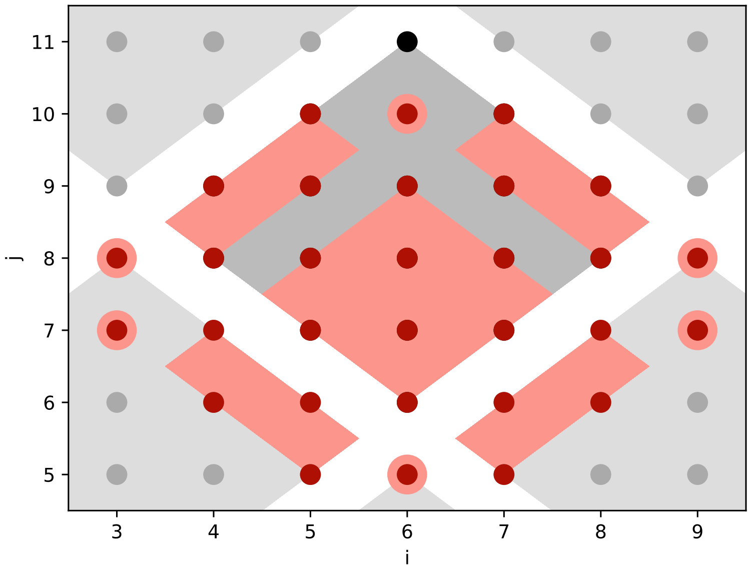


Figure 7.7: MARS for uniformly intersecting dependences (Jacobi 1D)

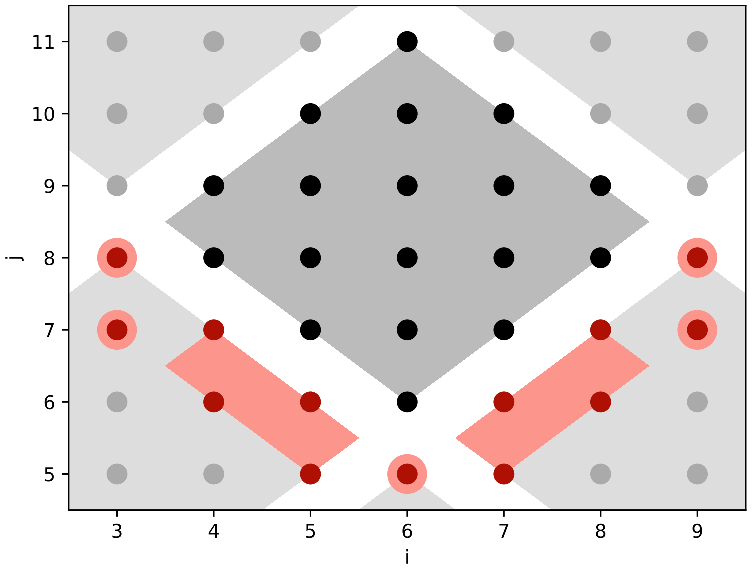


Figure 7.8: Coincidence between MARS computed with the uniform dependence method, and those computed with the affine method.

we need to write every dependence separately:

$$\forall i \in \{1, \ldots, Q\} : B_i(\vec{x}) = A_i\vec{x} + \vec{b}_i$$

and each dependence having its own null space, there is one orthonormal basis of the null space and a supplementary per dependence, and therefore one projection $W_i(\vec{t'} - \vec{t})$ per dependence.

Single null space requirement

Because the dependences may no longer have the same linear part, each linear part may have a different null space. When considering any consumer tile $\vec{t'} \in V(\vec{t})$, it is no longer true that the projection of $\vec{t'} - \vec{t}$ onto each null space is independent of the tile coordinates. The invariance by translation of a tile from Proposition 5 therefore no longer holds.

Figure 7.9 gives an example of this case with two dependences: $B_1(i, j) = (i - j)$ and $B_2(i, j) = (i + j)$. Here, the $\vec{p}$s depend on $\vec{t}$. Due to the dependence B_1, all the tiles northwest of each tile will intersect with its footprint; but the dependence B_2 generates a footprint southwest, also consumed (because of B_1) by all tiles to its northwest.

In Figure 7.9, we show the $W_1(\vec{t'} - \vec{t})$: $\ker(A_1)$ points to the northeast, and I_1 (supplementary of $\ker(A_1)$) points to the northwest, parallel to the dependence B_2.

A sufficient condition for a position-independent footprint to exist is that all dependences have the same null space:

Proposition 6. *If all dependences have the same null space, then all tiles have the same footprint up to a translation. Otherwise said, for any $\vec{\delta} \in \mathbf{Z}^t$, there exists $\vec{u} \in \mathsf{Im}(B)$ such that:*

For each $\vec{t} \in \mathbf{Z}^t$, if $\vec{t'} = \vec{t} + \vec{\delta}$, then

$$\bigcup_{i=1}^{Q} B_i \left\langle T(\vec{t'}) \right\rangle = \left\{ \vec{y} + \vec{u} : \vec{y} \in \bigcup_{i=1}^{Q} B_i \left\langle T(\vec{t}) \right\rangle \right\}$$

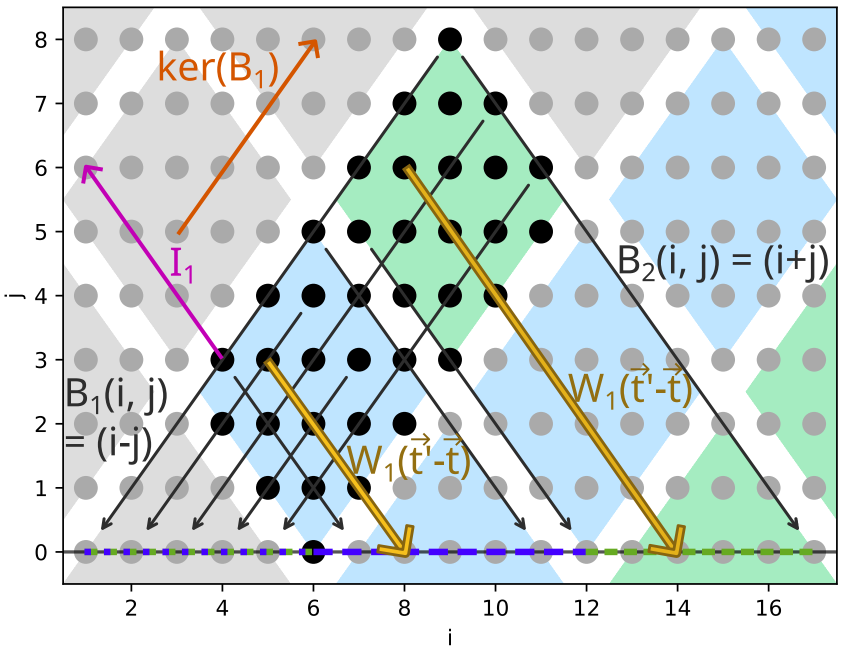


Figure 7.9: Non-uniformly intersecting dependences do not guarantee that the vectors $W_i(\vec{t'}-\vec{t})$ do not depend on $\vec{t}$, i.e. each tile's footprint is not necessarily a translation of another tile's footprint.

Proof. We know that the sought $\vec{u}$ exists for each dependence per Proposition 5: for each $i \in \{1, \dots, Q\}$, there is a $\vec{u}_i$ such that

$$B_i \left\langle T(\vec{t'}) \right\rangle = \left\{ \vec{y} + \vec{u}_i : \vec{y} \in B_i \left\langle T(\vec{t}) \right\rangle \right\}$$

This $\vec{u}_i$ is constructed as:

$$\vec{u}_i = \sum_{i=1}^{t} (t'_i - t_i) \vec{n}_i^p$$

where the $\vec{n}_i^p$s are the projections of the normal vectors onto a supplementary of the null space of each dependence. Because all of the dependences have the same null space, it comes that all of the $\vec{u}_i$ have the *same* projection onto the *same* supplementary of that null space. Therefore, they are all equal. □

Constructon of MARS with a single null space

We must prove the requirements stated in Section 7.4.2, proved in the previous two cases, still hold to compute the MARS.

The uniqueness of $\vec{p}$ (invariance by translation of a tile) has become a hypothesis, and the dependences must satisfy this requirement to compute MARS. The previous paragraph only gave a sufficient condition for it to be satisfied.

The finiteness (and enumerability) of the set representatives of consumer tiles still holds if the dependences all have the same null space.

We can construct the MARS using the same procedure as in 7.4.4: the footprints of all dependences are distinct, but the null space is the same, therefore the same definition for $P(\vec{t}) = \left\{W(\vec{t'} - \vec{t})\right\}$ as in 7.4.4 holds.

Case of multiple null spaces

If the dependences have multiple null spaces, there is no guarantee that MARS can be constructed (see 7.4.5 for a counter-example).

Due to the null spaces being different, there is no guarantee that:

- The consumer tiles which footprint intersects with that of $T(\vec{t})$ are located at uniform translations of $\vec{t}$ (see counter-example at 7.4.5), and
- The projections of translations from $\vec{t}$ to all other consumer tiles onto every null space are finite sets of vectors, i.e. the method used previously to obtain a finite set of representative consumer tiles still gives a finite set.

We propose a solution to the second point: we can obtain finite sets of translation vectors to represent all consumer tiles for all dependences, although these translations may be parametric.

The idea is to only consider the projection of dependences that contribute to a tile's footprint intersecting with $B\left\langle T(\vec{t})\right\rangle$; and make a partition of all the sets of consumer tiles according to the contributing dependences.

Indeed, it is equivalent to say that a dependence B_i contributes to the consumption, and that the consumer tile $\vec{t'}$ is in $V_i(\vec{t})$. Multiple dependences contribute to the consumption if and only if $\vec{t'}$ is simultaneously in all the $V_i(\vec{t})$s of these dependences (i.e. it is in their intersection).

Using this fact, we can partition the space of all consumer tiles according to which dependences contribute to each family. In other words, given $D \in \mathcal{P}(\{1, \ldots, Q\})$, we compute:

$$F_D(\vec{t}) = \bigcap_{i \in D} V_i(\vec{t}) \setminus \bigcup_{i \notin D} V_i(\vec{t})$$

Proposition 7. *Let* $\vec{t'} \in \bigcup_{i=1}^{Q} V_i(\vec{t})$. *There exists a unique* $D \in \mathcal{P}(\{1, \ldots, Q\})$ *(i.e. a unique* $D \subset \{1, \ldots, Q\}$*) such that* $\vec{t'} \in F_D$. *In other words,* $\{F_D : D \subset \{1, \ldots, Q\}\}$ *is a partition of the set of all consumer tiles* $\bigcup_{i=1}^{Q} V_i(\vec{t})$.

Proof. The construction of the $F_D(\vec{t})$s, given that $\{1, \ldots, Q\}$ is a finite set, guarantees the fact all the F_Ds are disjoint and therefore creates a partition of $\bigcup_{i=1}^{Q} V_i(\vec{t})$. □

For each $D \in \mathcal{P}(\{1, \ldots, Q\})$, $F_D(\vec{t})$ contains a family of consumer tiles (possibly empty). If it is not empty, then using the same reasoning as in Proposition 5 we can prove the following proposition:

Proposition 8. *Let* $D \in \mathcal{P}(\{1, \ldots, Q\})$. *Then:*

$$P_D(\vec{t}) = \left\{ W_i(\vec{t'} - \vec{t}) : \vec{t'} \in F_D \wedge i \in D \right\}$$

is a finite set.

Proof. Considering that $F_D(\vec{t}) \subset \bigcap_{i \in D} V_i(\vec{t})$, Proposition 5 can be applied to each individual $V_i(\vec{t})$. □

The effects of parametric vectors leading to consumer tiles are unknown at this point. Whether MARS can be constructed in this case is left as an open question.

7.4.6 Case of dependences between tiled spaces

Dependences that lead to tiled spaces correspond to the passing of intermediate results between tiles. These dependences, when uniform, were supported in Chapter 5, and transmission of intermediate results was done through MARS transiting in the main memory. This produced a partitioning of the *flow-out set* and *flow-in set* of each tile. In this section, we extend this principle to affine dependences.

Uniform dependences used in Chapter 5 guaranteed that the producer and consumer were in the same space (which is not the case with affine dependences), and the identity linear part of the dependences gave that the image of a tile by a dependence was a translation of the tile itself.

The main problem with having different consumer and producer spaces is the relation between the consumer tiles' "footprint" in the producer tiles' space, and the producer space tiling itself: the footprints of the consumer tiles by the dependences produce a tiling that may not match with the existing tiling of the producer space.

In the previous sections (7.4.3, 7.4.4, 7.4.5), the existence of MARS relied on the footprints of the consumer tiles (in a tiled iteration space) in the data space (hereafter *destination space*) being **independent of the consumer tile** (i.e. the origin of the dependence). In this section, the destination space is a tiled iteration space, and we want the tiling induced by the dependence to "match" the existing tiling or be finer than it. To this aim, we add the requirement is that the same footprints are **independent of the producer tile** (i.e. the destination of the dependence).

Assuming there are t tiling hyperplanes in the source space, and q tiling hyperplanes in the destination space, let their (unit) normal vectors be respectively $\vec{n}_1, \ldots, \vec{n}_t$ and $\vec{d}_1, \ldots, \vec{d}_q$ and their tile sizes $s_1, \ldots, s_t$ and $z_1, \ldots, z_q$. Let the scaled normal vectors (translation of one tile along each hyperplace) be $\vec{\mathbf{n}}_1, \ldots, \vec{\mathbf{n}}_t$ and $\vec{\mathbf{d}}_1, \ldots, \vec{\mathbf{d}}_q$.

Let there be Q dependences $B_1, \ldots, B_q$. Let $\vec{t} = (t_1, \ldots, t_t) \in \mathbf{CT}$ be a *consumer tile* vector coordinate, and let $\vec{u} = (u_1, \ldots, u_q) \in \mathbf{DT}$ be a *producer tile* vector coordinate (in the destination space of the dependences). Let a tile in the destination space be designated as $U(\vec{t})$ using a definition analogous to that of the source space (see 7.3). Let the consumer tiles of a destination tile $\vec{u} \in \mathbf{DT}$ be

$$X(\vec{u}) = \left\{\vec{t} \in \mathbf{CT} : \exists j \in \{1, \ldots, Q\} : B_j \left\langle T(\vec{t}) \right\rangle\right\}$$

and a tile translation vector in the destination space be expressed as:

$$N(\vec{u}) = \sum_{k=1}^{q} u_k \vec{\mathbf{d}}_k$$

The following proposition is a conjecture. It establishes the equivalence between a translation of a tile in the producer space and the translation of multiple tiles in the consumer space.

Proposition 9. *We have the following equivalence:*

$$\begin{gathered}
\forall \vec{u}, \vec{u'} \in \mathbf{DT}, \forall \vec{t} \in X(\vec{u}), \forall i \in \{1, \ldots, Q\} : \\
\exists \vec{t'} \in X(\vec{u'}) : B_i \left\langle T(\vec{t'}) \right\rangle = \left\{\vec{y} + N(\vec{u'} - \vec{u}) : \vec{y} \in B_i \left\langle T(\vec{t}) \right\rangle\right\} \\
\Updownarrow \\
\forall i \in \{1, \ldots, Q\}, \forall j \in \{1, \ldots, t\}, \forall k \in \{1, \ldots, q\} : \\
\exists m \in \mathbf{Z} : m((A_i \vec{\mathbf{n}}_j) \cdot \vec{d_k})\vec{d_k} = \vec{\mathbf{d}}_k
\end{gathered}$$

If proved, this proposition then establishes a condition on the dependences for there to be a unique control flow, independent of the tile coordinates, for both the MARS to produce (by each producer tile) and the MARS to retrieve (by each consumer tile).

7.5 Related work

This work introduces a partitioning of data arrays and iteration spaces based on the consumption pattern of each data. Existing work on partitioning aims at locality in the first place, before memory access optimization. Our work relies on a locality optimization (tiling) and seeks to further improve memory accesses. This work is made specific by the combination of its objective (partitioning data for spatial locality) and its method (fine-grain partitioning where iteration spaces are already partitioned).

7.5.1 Goal of partitioning

Existing work on partitioning mainly targets locality, such as Agarwal et al. [2]. Our work uses the same definitions and follows the same reasoning as this paper, with a different objective: while [2] seeks to adjust the tile size and shapes for locality (i.e. the footprint size of each tile), we seek to exhibit spatial locality (data contiguity) opportunities. In that sense, our work is not the first to propose a partitioning of iteration and data spaces using affine dependences; however, the desired result (with a spatial locality objective) differs, and so the construction procedure and hypotheses too.

Parallelism is also an objective: [88] perform iteration and data space partitioning, then fuse partitions to maximize computation parallelism while preserving locality. The resulting code is suitable for CPU and GPU implementation with a cache hierarchy; our partitioning scheme does not follow the temporal utilization of the retrieved data within a tile. It therefore is more adapted to scratchpad memories, and scenarios memory accesses can be decoupled from computations, because grouping data for spatial locality requires significant on-chip data movement. This makes our approach more suitable for task-level pipelined (*read, execute, write*) FPGA or ASIC accelerators, or for small CPU tiles (register tiling) where the register space can be considered a scratchpad.

7.5.2 Partitioning methods

Instead of partitioning the inter-tile communicated data with a tiling already known, one can consider partitioning the inputs and outputs, and deriving tiled iteration space tiling from the inputs or outputs themselves. This approach is taken in [88] where the tile shapes are iteratively constructed from the (tiled) consumers of the iterations or data.

Monoparametric tiling [40] is performed using an inverse approach as ours: the data spaces (variables) are partitioned into tiles, and then the iteration space is partitioned. It requires the program to be represented as a *system of affine recurrence equations*, where loops do not exist; instead, iteration spaces start to exist at code generation time, when a variable needs to be computed. The main difference is that our partitioning scheme must be applied after loop tiling, and therefore after most locality optimizations.

Dathathri et al. [7, 23] partitions the iteration spaces for inter-node communications in distributed systems, in a manner similar to MARS: the *flow-out* iterations of each tile are partitioned by dependences (*dependence polyhedra*) and consumer tiles (*receiving tiles*). While both approaches are similar with respect to how data is grouped and transmitted, ours is extended to data space partitioning. Our approach however adds a restriction on the dependences: we require that the flow-out partitions are invariant across all tiles, so that a simple, unique control flow can be derived. Our approach can then be used to create *position-independent* accelerators that can process any tile in the iteration space.

It is noteworthy that both our approach and [23], along with other domain-specific inter-node data partitioning schemes (e.g., [90]) acknowledge that, to achieve a high bandwidth utilization of the RAM or network, inter-tile (inter-node) communications need a specific data layout inferred using static analysis.

7.6 Conclusion

Optimizing programs with respect to memory accesses is a key to improving their performance. This chapter proposes an analysis method to automatically partition data and iteration spaces from the polyhedral representation of a program when loop tiling is applied.

Partitioning data arrays is already known to improve spatial locality and, in turn, access performance. In this chapter, we propose a fine-grain partitioning scheme that can be used to optimize spatial locality for both iteration and data spaces based on the polyhedral representation of a program. Although this chapter only proposes a theoretical study, the effects of partitioning the data spaces, e.g. with data tiling, are known to be beneficial; and likewise, Chapter 6 has shown that partitioning intermediate results and deriving a specific allocation yields performance improvements.

Implementing MARS for affine dependences would incur additional engineering than simply allocating intermediate results: the input data, usually intelligible to the user, would need to be transformed prior to execution; likewise, the inverse transformation would need to occur prior to handing back the results to the user. The profitability of partitioning the input data into MARS would depend on the time such a transformation would take.

Chapter 8

Conclusions and perspectives

This chapter summarizes and closes the work of this dissertation, and proposes future research directions.

8.1 Conclusions

The continued increase in parallelism exploitation and of compute power is driving programs towards the limits of current memory architectures in terms of bandwidth. To cater to the needs of each application, memory architectures are becoming more diverse, with a landscape of bandwidth, latency and capacity options. To fully exploit the bandwidth without suffering from the effects of latency, developers must adapt where the data is placed and how it is accessed. Manually creating data layouts is a complicated task because of the amount of program re-engineering it requires, and even more so if the program is a hardware design.

This dissertation addresses the issue for domain-specific hardware, by presenting automated methods to derive memory allocations from a program. It sets itself within a context where application-specific, ad-hoc memory allocations are sought, notably to relieve the memory-boundness of specific applications. This problem is especially important to tackle with intrinsically memory-bound applications, such as matrix-vector products [26].

This dissertation shows that it is possible to exhibit contiguous memory layouts and access patterns from the polyhedral representation of a program; in this respect, it proposes an allocation for accelerators using high-bandwidth memories seeking the longest contiguous accesses (Chapter 4), and an irredundant allocation for hardware accelerators seeking to maximize the usefulness of the data on the bus while keeping as much contiguous accesses as possible (Chapter 6).

A second contribution brought in by this dissertation is an automation aspect: both allocations proposed are accompanied by algorithms and methodologies to automatically allocate the data and create access functions. This makes it possible to create compiler passes transparently implementing the proposed allocations. Chapters 4 and 5 describe the mechanisms such compiler passes may implement to generate the memory layouts and access functions.

A third contribution of this dissertation, used as a core element to build memory allocations, are partitionings of the iteration and data spaces of polyhedral programs. While we only suggest a use case for memory allocation, these partitionings can be used for other purposes such as fault tolerance. The partitioning methods are based on the program's dependences, and two cases are studied: Chapter 5 covers the case of uniform dependences, and Chapter 7 covers the broader case of affine dependences.

8.2 Perspectives

The work brought in by this dissertation consists in memory allocations for domain-specific accelerators, and supporting polyhedral compilation analysis and transformation techniques. These works are very specific, yet leave unexplored areas and open perspectives on multiple fronts. We finish this dissertation by listing a few of them.

8.2.1 On the polyhedral compilation aspect

In this dissertation, we have studied memory allocations for simple cases of tiling, where the tiling hyperplanes verify certain hypotheses (e.g. they are canonical in Chapter 4, linearly independent from each other in Chapter 7). Further extending it to relax these hypotheses may enable more programs to benefit from the memory allocations presented in this work.

Prior work, e.g. [92], has shown overlapped tiling brings performance benefits due to additional parallelism at the expense of redundant computations. Our work currently does not support such tiles, where the data flow analysis would need to take into account the redundant computations.

A refined definition of tiles and tiling hyperplanes could lead to the support of tiling schemes with non-linearly independent hyperplanes, or with specific cuts such as diamond tiling [9], monoparametric tiling [40] or fancier tile shapes such as [91].

8.2.2 On using more hardware properties

The work of this dissertation focuses on finding data flowing from tile to tile and generating allocation functions. Chapters 5 and 7 propose to partition the data according to the consumption pattern, into coarse-grain blocks. This results in block-level allocations, where the layout within each block of data is not specified. Using information on the target architecture, it is possible to tune the inner layout of such blocks to relieve bank or port contention. This is especially true when the data is partitioned into banks (e.g. into Block RAMs inside FPGA chips, or in HBM stacks).

Another aspect is minimizing the access latency in the global memory. Optimizing the layout of the MARS themselves as is done in Chapter 6 is done without respect to the structure of the off-chip memory. The access latency of that external memory contributes to the total access latency, and it therefore should be minimized. When DDR memories are used, access latency is for instance lower for data in different banks than in different rows of the same bank. Knowing such internals of the memory, other placements of MARS minimizing the access latency while preserving contiguity (and therefore a low number of transactions) could further improve performance.

8.2.3 On use cases of polyhedral-based allocations

The proposed methods mainly target domain-specific hardware architectures that manange memory accesses on their own. However, we can suggest they be extended to more generic architectures, and not necessarily for bandwidth optimization. Data contiguity can, for instance, enable automatic vectorization, and contribute to a better utilization of vector units with fewer on-chip, register-to-register movements. Using MARS on register tiles [42] would be another angle of attack to a register allocation problem tackled in domain-specific optimizers such as YASK [84].

Publications and Contributions

The publications listed here are those effective at the date this manuscript is produced, and exclude those papers currently under review.

Journal articles

Corentin Ferry, Tomofumi Yuki, Steven Derrien, and Sanjay Rajopadhye. "Increasing FPGA Accelerators Memory Bandwidth with a Burst-Friendly Memory Layout". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022), pp. 1–1. DOI: 10.1109/tcad.2022.3201494

Workshop papers

Corentin Ferry, Steven Derrien, and Sanjay Rajopadhye. "Maximal Atomic irRedundant Sets: a Usage-based Dataflow Partitioning Algorithm". In: *13th International Workshop on Polyhedral Compilation Techniques (IMPACT'23)*. 2023. URL: https://impact-workshop.org/impact2023/papers/paper1.pdf

Corentin Ferry, Steven Derrien, and Sanjay Rajopadhye. "An Irredundant Decomposition of Data Flow with Affine Dependences". In: *14th International Workshop on Polyhedral Compilation Techniques (IMPACT'24)*. 2024. URL: https://impact-workshop.org/impact2024/papers/paper7.pdf

Presentations

Corentin Ferry, Steven Derrien, Sanjay Rajopadhye, and Tomofumi Yuki. *Canonical Facet Allocation: A Burst-Friendly Data Layout.* GDR SoC (Poster). 2021

Bibliography

[1] Thea Aarrestad et al. “Fast convolutional neural networks on FPGAs with hls4ml”. In: *Machine Learning: Science and Technology* 2.4 (July 2021), p. 045015. DOI: 10.1088/2632-2153/ac0ea1.

[2] A. Agarwal, D.A. Kranz, and V. Natarajan. “Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors”. In: *IEEE Transactions on Parallel and Distributed Systems* 6.9 (1995), pp. 943–962. DOI: 10.1109/71.466632.

[3] C. Bastoul. “Code Generation in the Polyhedral Model Is Easier Than You Think”. In: *PACT’13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, Sept. 2004, pp. 7–16.

[4] Cédric Bastoul et al. “Putting Polyhedral Loop Transformations to Work”. In: *LCPC’16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*. College Station, Texas, Oct. 2003, pp. 209–225.

[5] Samuel Bayliss and George A. Constantinides. “Optimizing SDRAM bandwidth for custom FPGA loop accelerators”. In: *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA ’12*. ACM Press, 2012. DOI: 10.1145/2145694.2145727.

[6] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. “Automatic Intra-Register Vectorization for the Intel Architecture”. In: *International Journal of Parallel Programming* 30.2 (2002), pp. 65–98. DOI: 10.1023/a:1014230429447.

[7] Uday Bondhugula. “Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2013. DOI: 10.1145/2503210.2503289.

[8] Uday Bondhugula. "Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model". PhD thesis. Ohio State University, 2008. URL: https://www.csa.iisc.ac.in/~udayb/publications/uday-thesis.pdf.

[9] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. "Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations". In: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (May 2017), pp. 1285–1298. DOI: 10.1109/tpds.2016.2615094.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 101–113. ISBN: 9781595938602. DOI: 10.1145/1375581.1375595.

[11] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Program Optimization System". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* June 2008. DOI: 10.1145/1379022.1375595.

[12] Uday Bondhugula et al. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *International Conference on Compiler Construction (ETAPS CC).* Apr. 2008. DOI: 10.1007/978-3-540-78791-4_9.

[13] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. "SODA: Stencil with optimized dataflow architecture". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE. ACM, Nov. 2018, pp. 1–8. DOI: 10.1145/3240765.3240850.

[14] Young-kyu Choi and Jason Cong. “HLS-based optimization and design space exploration for applications with variable loop bounds”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[15] Young-Kyu Choi et al. “In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms”. In: *ACM Transactions on Reconfigurable Technology and Systems* 12.1 (Feb. 2019), pp. 1–20. DOI: 10.1145/3294054.

[16] Stephanie Coleman and Kathryn S. McKinley. “Tile Size Selection Using Cache Organization and Data Layout”. In: *SIGPLAN Not.* 30.6 (June 1995), pp. 279–290. ISSN: 0362-1340. DOI: 10.1145/223428.207162.

[17] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. “Automatic memory partitioning and scheduling for throughput and power optimization”. In: *ACM Transactions on Design Automation of Electronic Systems, Vol. 16, No. 2, Article 1* 16 (2011), pp. 1–25. ISSN: 1084-4309. DOI: 10.1145/1929943.1929947.

[18] Jason Cong et al. “Source-to-source optimization for HLS”. In: *FPGAs for Software Programmers* (2016), pp. 137–163.

[19] Jason Cong et al. “Understanding Performance Differences of FPGAs and GPUs”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Apr. 2018. DOI: 10.1109/fccm.2018.00023.

[20] Gabor Csordas, Mikhail Asiatici, and Paolo Ienne. “In Search of Lost Bandwidth: Extensive Reordering of DRAM Accesses on FPGA”. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, Dec. 2019. DOI: 10.1109/icfpt47387.2019.00030.

[21] P Cummiskey, Nikil S. Jayant, and James L. Flanagan. “Adaptive Quantization in Differential PCM Coding of Speech”. In: *Bell System Technical Journal* 52 (Sept. 1973). DOI: 10.1002/j.1538-7305.1973.tb02007.x.

[22] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. “Extended lattice-based memory allocation”. In: *Proceedings of the 25th International Conference on Compiler Construction*. CGO ’16. ACM, Mar. 2016. DOI: 10.1145/2892208.2892213.

[23] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. “Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-Memory”. In: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Oct. 2013. DOI: 10.1109/PACT.2013.6618833.

[24] Gaël Deest. “Implementation trade-offs for FGPA accelerators”. PhD thesis. Université de Rennes 1, 2017.

[25] Gael Deest, Tomofumi Yuki, Sanjay Rajopadhye, and Steven Derrien. “One size does not fit all: Implementation trade-offs for iterative stencil computations on FPGAs”. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2017. DOI: 10.23919/fpl.2017.8056781.

[26] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. “High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS”. In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Feb. 2022. DOI: 10.1145/3490422.3502368.

[27] Paul Feautrier. “Dataflow analysis of array and scalar references”. In: *International Journal of Parallel Programming* 20.1 (Feb. 1991), pp. 23–53. DOI: 10.1007/BF01407931.

[28] Corentin Ferry, Steven Derrien, and Sanjay Rajopadhye. “An Irredundant Decomposition of Data Flow with Affine Dependences”. In: *14th International Workshop on Polyhedral Compilation Techniques (IMPACT’24)*. 2024. URL: https://impact-workshop.org/impact2024/papers/paper7.pdf.

[29] Corentin Ferry, Steven Derrien, and Sanjay Rajopadhye. “Maximal Atomic irRedundant Sets: a Usage-based Dataflow Partitioning Algorithm”. In: *13th International*

Workshop on Polyhedral Compilation Techniques (IMPACT'23). 2023. URL: https://impact-workshop.org/impact2023/papers/paper1.pdf.

[30] Corentin Ferry, Steven Derrien, Sanjay Rajopadhye, and Tomofumi Yuki. *Canonical Facet Allocation: A Burst-Friendly Data Layout*. GDR SoC (Poster). 2021.

[31] Corentin Ferry, Tomofumi Yuki, Steven Derrien, and Sanjay Rajopadhye. "Increasing FPGA Accelerators Memory Bandwidth with a Burst-Friendly Memory Layout". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022), pp. 1–1. DOI: 10.1109/tcad.2022.3201494.

[32] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. "Cache miss equations". In: *ACM Transactions on Programming Languages and Systems* 21.4 (July 1999), pp. 703–746. DOI: 10.1145/325478.325479.

[33] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly — Performing Polyhedral Optimizations On A Low-Level Intermediate Representation". In: *Parallel Processing Letters* 22.04 (Dec. 2012), p. 1250010. DOI: 10.1142/s0129626412500107.

[34] Yijin Guan et al. "Using Data Compression for Optimizing FPGA-Based Convolutional Neural Network Accelerators". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2017, pp. 14–26. DOI: 10.1007/978-3-319-67952-5_2.

[35] José R. Herrero and Juan J. Navarro. "Using Non-canonical Array Layouts in Dense Matrix Operations". In: *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer Berlin Heidelberg, 2006, pp. 580–588. DOI: 10.1007/978-3-540-75755-9_70.

[36] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. "High-Performance Code Generation for Stencil Computations on GPU Architectures". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: Association for Computing Machinery, 2012, pp. 311–320. ISBN: 9781450313162. DOI: 10.1145/2304576.2304619.

[37] Changwan Hong et al. “Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses”. In: *SIGPLAN Not.* 51.6 (June 2016), pp. 129–144. ISSN: 0362-1340. DOI: 10.1145/2980983.2908123.

[38] Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. “Custom High-Performance Vector Code Generation for Data-Specific Sparse Computations”. In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT ’22. Chicago, Illinois: Association for Computing Machinery, 2023, pp. 160–171. ISBN: 9781450398688. DOI: 10.1145/3559009.3569668. URL: https://doi.org/10.1145/3559009.3569668.

[39] Kuang-Hua Huang and Jacob A. Abraham. “Algorithm-Based Fault Tolerance for Matrix Operations”. In: *IEEE Transactions on Computers* C-33.6 (June 1984), pp. 518–528. DOI: 10.1109/tc.1984.1676475.

[40] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. “Monoparametric Tiling of Polyhedral Programs”. In: *International Journal of Parallel Programming* 49.3 (Mar. 2021), pp. 376–409. DOI: 10.1007/s10766-021-00694-2.

[41] F. Irigoin and R. Triolet. “Supernode partitioning”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’88*. ACM. ACM Press, Jan. 1988, pp. 319–328. DOI: 10.1145/73560.73588.

[42] M. Jiménez, J. M. Llabería, A. Fernández, and E. Morancho. “A general algorithm for tiling the register level”. In: *Proceedings of the 12th international conference on Supercomputing*. ACM, July 1998. DOI: 10.1145/277830.277859.

[43] Marta Jiménez, José M. Llabería, and Agustín Fernández. “Register tiling in nonrectangular iteration spaces”. In: *ACM Transactions on Programming Languages and Systems* 24.4 (July 2002), pp. 409–453. DOI: 10.1145/567097.567101.

[44] Lana Josipović, Radhika Ghosal, and Paolo Ienne. "Dynamically Scheduled High-level Synthesis". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Feb. 2018. DOI: 10.1145/3174243.3174264.

[45] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. "The cache performance and optimizations of blocked algorithms". In: 25 (1991), pp. 63–74. ISSN: 0163-5980. DOI: 10.1145/106974.106981.

[46] Samuel Larsen and Saman Amarasinghe. "Exploiting superword level parallelism with multimedia instruction sets". In: *ACM SIGPLAN Notices* 35.5 (May 2000), pp. 145–156. DOI: 10.1145/358438.349320.

[47] Hervé Le Verge. "Un environnement de transformations de programmes pour la synthese d'architectures regulières". PhD thesis. Université de Rennes 1, 1992.

[48] Shiqing Li, Di Liu, and Weichen Liu. "Efficient FPGA-based Sparse Matrix-Vector Multiplication with Data Reuse-aware Compression". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023), pp. 1–1. DOI: 10.1109/tcad.2023.3281715.

[49] Jun Liu et al. "A compiler framework for extracting superword level parallelism". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2012. DOI: 10.1145/2254064.2254106.

[50] Junyi Liu, Samuel Bayliss, and George A Constantinides. "Offline synthesis of online dependence testing: Parametric loop pipelining for HLS". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2015, pp. 159–162.

[51] Junyi Liu, John Wickerson, Samuel Bayliss, and George A Constantinides. "Polyhedral-based dynamic loop pipelining for high-level synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.9 (2017), pp. 1802–1815.

[52] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. “Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking”. In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Feb. 2021. DOI: 10.1145/3431920.3439284.

[53] John Lu and Keith D. Cooper. “Register promotion in C programs”. In: *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation - PLDI ’97*. ACM Press, 1997. DOI: 10.1145/258915.258943.

[54] Daniel Maier, Biagio Cosenza, and Ben Juurlink. “Local memory-aware kernel perforation”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, Feb. 2018. DOI: 10.1145/3168814.

[55] Christophe Mauras. “Alpha : un langage équationnel pour la conception et la programmation d’architectures parallèles synchrones”. PhD thesis. Université de Rennes 1, 1989.

[56] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. “Tile Size Selection Revisited”. In: *ACM Trans. Archit. Code Optim.* 10.4 (Dec. 2013). ISSN: 1544-3566. DOI: 10.1145/2541228.2555292.

[57] Hiroki Nakahara, Zhiqiang Que, and Wayne Luk. “High-Throughput Convolutional Neural Network on an FPGA by Customized JPEG Compression”. In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, May 2020. DOI: 10.1109/fccm48280.2020.00010.

[58] O. Ozturk, M. Kandemir, and M.J. Irwin. “Using Data Compression for Increasing Memory System Utilization”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.6 (June 2009), pp. 901–914. DOI: 10.1109/tcad.2009.2017430.

[59] S. Pillai and M.F. Jacome. “Compiler-directed ILP extraction for clustered VLIW/EPIC machines: predication, speculation and modulo scheduling”. In: *2003 Design, Automa-*

tion and Test in Europe Conference and Exhibition. IEEE Comput. Soc, 2003. DOI: 10.1109/DATE.2003.1253646.

[60] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. "Polyhedral-based data reuse optimization for configurable computing". In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '13*. ACM Press, 2013. DOI: 10.1145/2435264.2435273.

[61] Louis-Noël Pouchet and Tomofumi Yuki. *PolyBench/C 4.2.1*. 2016. URL: http://polybench.sf.net.

[62] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*. ACM Press, 2013. DOI: 10.1145/2491956.2462176.

[63] J. Ramanujam and P. Sadayappan. "Tiling multidimensional iteration spaces for multicomputers". In: *Journal of Parallel and Distributed Computing* 16.2 (Oct. 1992), pp. 108–120. DOI: 10.1016/0743-7315(92)90027-k.

[64] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout, and Sanjay Rajopadhye. "Parameterized loop tiling". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34.1 (2012), p. 3. DOI: 10.1145/2160910.2160912.

[65] Gabriel Rivera and Chau-Wen Tseng. "Data transformations for eliminating conflict misses". In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*. ACM Press, 1998. DOI: 10.1145/277650.277661.

[66] Tiago Santos and João MP Cardoso. "Automatic selection and insertion of hls directives via a source-to-source compiler". In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2020, pp. 227–232.

[67] Somayeh Sardashti, Andre Seznec, and David A. Wood. "Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache". In: *ACM Trans. Archit. Code Optim.* 13.3 (Sept. 2016), 27:1–27:25. ISSN: 1544-3566. DOI: 10.1145/2976740. URL: http://doi.acm.org/10.1145/2976740.

[68] Robert Schreiber and Jack J. Dongarra. *Automatic blocking of nested loops*. Tech. rep. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990.

[69] Jun Shirako and Vivek Sarkar. "An Affine Scheduling Framework for Integrating Data Layout and Loop Transformations". In: *Languages and Compilers for Parallel Computing*. Springer International Publishing, 2022, pp. 3–19. DOI: 10.1007/978-3-030-95953-1_1.

[70] A. Skodras, C. Christopoulos, and T. Ebrahimi. "The JPEG 2000 still image compression standard". In: *IEEE Signal Processing Magazine* 18.5 (2001), pp. 36–58. DOI: 10.1109/79.952804.

[71] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. "AutoDSE: Enabling software programmers to design efficient FPGA accelerators". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27.4 (2022), pp. 1–27.

[72] Gongjin Sun, Seongyoung Kang, and Sang-Woo Jun. "BurstZ+: Eliminating The Communication Bottleneck of Scientific Computing Accelerators via Accelerated Compression". In: *ACM Transactions on Reconfigurable Technology and Systems* 15.2 (June 2022), pp. 1–34. DOI: 10.1145/3476831.

[73] Teng Tian et al. "Exploration of Memory Access Optimization for FPGA-based 3D CNN Accelerator". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2020, pp. 1650–1655. DOI: 10.23919/DATE48585.2020.9116376.

[74] Sven Verdoolaege. “isl: An Integer Set Library for the Polyhedral Model”. In: *Mathematical Software – ICMS 2010*. Springer Berlin Heidelberg, 2010, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

[75] Sven Verdoolaege and Tobias Grosser. “Polyhedral Extraction Tool”. In: *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France, Jan. 2012.

[76] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. “The ALPHA language and its use for the design of systolic arrays”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 3.3 (Sept. 1991), pp. 173–182. DOI: 10.1007/bf00925828.

[77] Xuechao Wei, Yun Liang, and Jason Cong. “Overcoming Data Transfer Bottlenecks in FPGA-based DNN Accelerators via Layer Conscious Memory Management”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, June 2019. DOI: 10.1145/3316781.3317875.

[78] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Communications of the ACM* 52.4 (Apr. 2009), p. 65. DOI: 10.1145/1498765.1498785.

[79] Michael E. Wolf and Monica S. Lam. “A data locality optimizing algorithm”. In: *ACM SIGPLAN Notices* 26.6 (June 1991), pp. 30–44. ISSN: 0362-1340. DOI: 10.1145/113446.113449.

[80] David Wonnacott. “Time Skewing for Parallel Computers”. In: *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2000, pp. 477–480. DOI: 10.1007/3-540-44905-1_35.

[81] Shaojie Xiang et al. “HeteroFlow: Ac Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs”. In: *International Symposium on Field-*

Programmable Gate Arrays (FPGA). 2022. DOI: 10.1145/3490422.3502369. URL: https://www.csl.cornell.edu/~zhiruz/pdfs/heteroflow-fpga2022.pdf.

[82] Hanchen Ye et al. "ScaleHLS". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. ACM, July 2022. DOI: 10.1145/3489517.3530631.

[83] Charles Yount. "Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation". In: (Aug. 2015). DOI: 10.1109/hpcc-css-icess.2015.27.

[84] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. "YASK—Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning". In: *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE, Nov. 2016. DOI: 10.1109/wolfhpc.2016.08.

[85] Tomofumi Yuki et al. "Alphaz: A system for design space exploration in the polyhedral model". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2012, pp. 17–31.

[86] Chen Zhang et al. "Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.11 (Nov. 2019), pp. 2072–2085. DOI: 10.1109/TCAD.2017.2785257.

[87] Chen Zhang et al. "Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060.

[88] Jie Zhao and Peng Di. "Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data". In: *2020 53rd Annual IEEE/ACM In-*

ternational Symposium on Microarchitecture (MICRO). IEEE, Oct. 2020. DOI: 10.1109/micro50266.2020.00044.

[89] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A Constantinides. "POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations". In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2022, pp. 235–242.

[90] Tuowen Zhao, Mary Hall, Hans Johansen, and Samuel Williams. "Improving communication by optimizing on-node data movement with data layout". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, Feb. 2021. DOI: 10.1145/3437801.3441598.

[91] Tuowen Zhao et al. "Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2019. DOI: 10.1145/3295500.3356210.

[92] Xing Zhou et al. "Hierarchical overlapped tiling". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, Mar. 2012. DOI: 10.1145/2259016.2259044.