POLITECNICO DI MILANO
DEPARTMENT OF INFORMATION TECHNOLOGY
DOCTORAL PROGRAMME IN COMPUTER SCIENCE AND ENGINEERING

ON THE ROLE OF POLYHEDRAL ANALYSIS IN HIGH PERFORMANCE RECONFIGURABLE HARDWARE BASED COMPUTING SYSTEMS

Doctoral Dissertation of:
Riccardo Cattaneo

Advisor:
Prof. Marco D. Santambrogio

Tutor:
Prof. Donatella Sciuto

The Chair of the Doctoral Program:
Prof. Carlo Fiorini

2015 – XVIII

Acknowledgements

Summary

RECENT years have seen dramatic improvements in High Level Synthesis (HLS) tools: from efficient translation of numeric algorithms to image processing algorithms, from stencil computations to neural networks, relevant application domains and industries are benefiting from research on compute and/or memory/-communications network.

In order to systematically synthesize better circuits for specific programs and kernels, last decades' studies focused on the development of sound, formal approaches; one notable such framework is the Polyhedral Model (PM) and the associated code analysis technique, collectively called Polyhedral Analysis (PA). Under this representation it is possible to compute dependencies, find loop bounds, and reorder instructions in a completely automated manner relying on the same set of sound and comprehensive assumptions of PM.

We reconsider this rich state of the art, and elaborate on different methodologies and implement the related toolchains to generate highly parallel and power efficient kernels running on reconfigurable hardware, with the aim of distributing the workload on multiple computational blocks while maximizing the overall power efficiency of the resulting heterogeneous system. We approach this problem in three different, interrelated ways.

First of all, we dedicated our early efforts to the development of an accelerator-rich platform where the focus is on the coordination of multiple custom and software processors, via a novel Do-

main Space Exploration (DSE) phase. Specifically, the work elaborates on two relevant aspects: the effectiveness of Partial Reconfiguration (PR) to attain improved energy delay and throughput metrics, and the effectiveness of the heuristics chosen to realize the DSE step, which feature both low complexity and good exploration times. To this matter, we devote **Chapter 2**. We also extended the scope of this work by assuming that multiple computing elements are in place, and an adequate communication architecture is required to coordinate those accelerators. This is discussed in **Chapter 3**.

Secondly, we focus on amply data-parallel codes, and develop a novel HLS approach to using PM as a means to explicitly extract and isolate data and computation from affine codes in order to efficiently divide the workload among an arbitrary number of nodes, in the light of the current and foreseeable trend of adoption of reconfigurable hardware in the datacenter; towards energy proportional computing, we improve the current state of art in single core acceleration, as our methodology obtains near-linear speedup with the area at disposition to accelerate the given workload. To this subject, we devote **Chapter 4 and 5**.

Lastly, we focus on a specific, and more restricted class of data parallel codes, namely Iterative Stencil Loop (ISL), as they play a crucial role in a variety of different fields of application. The computationally intensive nature of those algorithms created the need for solutions to efficiently implement them in order to save both execution time and energy. This, in combination with their regular structure, has justified their widespread study and the proposal of largely different approaches to their optimization. However, most of these works are focused on aggressive compile time optimization, cache locality optimization, and parallelism extraction for the multicore/multi processor domain, while fewer works are focused on the exploitation of custom architectures to further exploit the regular structure of ISLs, specifically with the goal of improving power efficiency. This work introduces a methodology to systematically design power efficient hardware accelerators for the optimal execution of ISLs algorithms on Field Programmable Gate Arrays (FPGAs). To this extensive methodology, we devote **Chapter 6**.

As part of the methodology, we introduce the notion of Streaming Stencil Time-step (SST), a *streaming-based architecture* capable of achieving both low resource usage and efficient data reuse thanks to an *optimal* data buffering strategy; and we introduce a technique,

called SSTs *queuing*, capable to deliver a *quasi-linear* execution time speedup with constant bandwidth. This is the core subject of **Chapter 7**.

We validate all methodologies, approaches, and toolchains on significant benchmarks on either a Zynq-7000 or Virtex-7 FPGA using the *Xilinx Vivado* suite. Results, which are reported in each Chapter devoted to the respective subject, demonstrate how we are able to improve the efficiency of all the baselines we compare against; specifically, we improve the energy delay and throughput metric when using our accelerator-rich platform, and dramatically improve the on-chip memory resources usage using the PA-based methodologies, allowing us to treat problem sizes whose implementation would otherwise not be possible via direct synthesis of the original, unmanipulated HLS code.

We finally conclude the dissertation in **Chapter 8** with a number of potential future works.

Sommario

DURANTE gli ultimi anni gli strumenti di sintesi ad alto livello (*High Level Synthesis (HLS)*) hanno conosciuto notevoli miglioramenti: dalla traduzione efficiente di algoritmi numerici a quella degli algoritmi di elaborazione delle immagini, dai calcoli stencil alle reti neurali, disparati domini applicativi ed industrie stanno beneficiando della ricerca sulla sintesi di elementi di calcolo, sottosistemi di memoria, e reti di comunicazione.

La ricerca degli ultimi decenni si é focalizzata sullo sviluppo di approcci formali e corretti finalizzati alla sistematica sintesi di circuiti piú performanti per specifici programmi e nuclei computazionali complessi. In tale senso é di notevole interesse il Modello Poliedrale (*Polyhedral Model (PM)*) e la tecnica di analisi codice associata, cui ci si riferisce collettivamente con il termine Analisi Poliedrale (*Polyhedral Analysis (PA)*), descritta con dovizia di particolari nel capitolo 4. Tramite questa rappresentazione e manipolazione della computazione é possibile calcolare le dipendenze, i gli indici degli estremi delle iterazioni, riordinare le istruzioni in modo completamente automatizzato, ed altro ancora. In questa tesi riconsideriamo lo stato dell'arte ed i lavori relativi a quest'area di ricerca, elaborando metodologie innovative ed implementando le relative toolchain per generare hardware riconfigurabile, con l'obiettivo di distribuire il carico di lavoro su piú blocchi di calcolo, massimizzando l'efficienza energetica complessiva di il sistema eterogeneo risultante. Approcciamo il problema in tre modi differenti.

Prima di tutto, abbiamo sviluppato una piattaforma basata su acceleratori in cui l'attenzione é rivolta al coordinamento di piú processori e software personalizzati, attraverso una fase di esplorazione dello spazio dei parametri (*Domain Space Exploration (DSE)*). Il lavoro approfondisce due aspetti rilevanti: primo, l'efficacia della riconfigurazione parziale (*Partial Reconfiguration (PR)*) per ottenere un miglioramento delle metriche prodotto energia-latenza e throughput; secondo, l'efficacia delle euristiche scelte per realizzare il passo DSE, di bassa complessitá computazionale e con ottimi tempi di esecuzione. A questo lavoro dedichiamo il capitolo 2. Abbiamo anche ampliato la portata dello stesso assumendo che piú elementi di calcolo siano presenti nel sistema, e sia necessario un'adeguata architettura di comunicazione al fine di coordinare tali acceleratori. Questo é discusso nel capitolo 3.

Successivamente ci concentriamo sui codici affini ampiamente paralleli, e sviluppiamo un nuovo approccio alla sintesi ad alto livello (HLS) per utilizzare l'analisi poliedrale come un mezzo per estrarre ed isolare flusso dati e computazione, per poter dividere in modo efficiente il carico di lavoro tra un numero arbitrario di nodi alla luce della tendenza attuale e futura di adozione di hardware riconfigurabile nei centri di elaborazione dati; nell'ottica dei sistemi di calcolo *energy proportional*, miglioriamo l'attuale stato dell'arte in termini di accelerazione single core, come dimostra la metodologia descritta nel capitolo 5.

Infine, ci concentriamo su una classe specifica e ristretta di codici paralleli, gli *iterative Stencil Loop (ISL)*, codici la cui computazione efficiente risulta cruciale in una varietá di campi di applicazione. La natura computazionalmente intensiva di questi algoritmi ha generato la necessitá di disporre di metodi efficienti per la loro computazione al fine di risparmiare sia tempo che energia per la loro esecuzione. Questo, in combinazione con la loro struttura regolare, ha giustificato il loro ampio studio e la proposta di molteplici approcci per loro ottimizzazione. Tuttavia, la maggior parte di queste opere sono focalizzate sull'ottimizzazione statica aggressiva, l'ottimizzazione della localitá dei dati, e l'estrazione di parallelismo nel dominio multicore, mentre un minor numero di lavori sono focalizzati sullo sfruttamento di architetture riconfigurabili con l'obiettivo di migliorare l'efficienza energetica della computazione. Come parte della metodologia, si introduce il concetto di *Streaming Stencil Time-step (SST)*, un'architettura basata sul modello *dataflow* in grado di coniugare tanto un basso uti-

lizzo delle risorse quanto un riuso efficiente dei dati grazie ad una strategia ottimale di buffering dei dati; e si introduce una tecnica, chiamata accodamento di SST, in grado di fornire una riduzione del tempo di esecuzione in funzione del numero di elementi di calcolo presenti, con larghezza di banda *off-chip* costante, e incremento di efficienza energetica fino a saturazione del sistema di calcolo. I capitoli 6 e 7 introducono questa metodologia per la progettazione sistematica di acceleratori hardware efficienti per l'esecuzione ottimale di algoritmi ISL su *Field Programmable Gate Array (FPGA)* mediante SST.

Convalidiamo ogni metodo, approccio, e toolchain su applicazioni di riferimento implementate su Zynq-7000 o Virtex-7 utilizzando la suite Xilinx Vivado. I risultati, che sono riportati in ogni capitolo dedicato al rispettivo argomento, dimostrano come siamo in grado di migliorare l'efficienza di tutte le soluzioni di base con cui ci confrontiamo; in particolare, miglioriamo il prodotto ritardo-energia ed il throughput quando utilizziamo la nostra piattaforma basata su acceleratori; miglioriamo sensibilmente l'utilizzo di risorse *on-chip* di memoria utilizzando le metodologie basate su PA, cosa che ci permette di trattare problemi dimensioni la cui attuazione sarebbe altrimenti non essere possibile via diretta sintesi del codice HLS manipolata originale; e siamo in grado di scalare la computazione basata su SST secondo modello teorico, cosa che massimizza l'efficienza energetica del sistema di calcolo risultante.

Concludiamo la tesi col capitolo 8, in cui forniamo spunti per eventuali sviluppi futuri di queste linee di ricerca.

Contents

Contents

CHAPTER 1

Introduction

In this Chapter it is introduced the context required to motivate the work done in this thesis. Section 1.1 describes this context, namely the role of reconfigurable hardware in the advancement of High Performance Computing (HPC) systems. In Section 1.2 the main challenges that arise when designing the next generation systems are presented, while Section 1.3 provides a brief description of the heterogeneous systems, with particular attention to Field Programmable Gate Arrays (FPGAs). The particular role in computing of these devices is described in Section 1.4. Finally, Section 1.5 provides a high level overview of the proposed work within the context of building more power efficient computing systems. Finally, Section 1.5 delineates the contributions and the structure of subsequent chapters of this thesis.

1.1 Context

The next generation of super computing platforms will deliver unprecedented performance for the benefit of all scientific disciplines

and industries.

In HPC, the important milestones are considered the emergence of systems whose overall performance, expressed as the number of Floating Point Operations Per Second (FLOPS) a given system is able to perform, crosses the threshold of 10^{3k}, for some $k \in \mathbb{N}$. A first important achievement was made in 1985 where the Gigascale (10^9) was reached with the *Cray-2*. In 1997 Terascale (10^{12}) was delivered by Intel's *ASCI Red*, and in 2008 Petascale (10^{15}) was achieved by the IBM's *Roadrunner*. It is believed that in the near future, approximately in 2020, top tier systems will perform at least at the Exascale (10^{18}) level.

The need for such technological advancements is easily justified: some of the key computational challenges that are faced not only by industry and science, but our civilization as a whole, can only be addressed using more and more powerful computing systems. There are a lot of practical problems that can benefit substantially from it: in climate modeling, it could help to adapt faster to climate changes and sea level rise thanks to a much more accurate forecasting; in medical systems it could allow a dramatic advancement in the research for preventing and curing cancer as well as the other challenging diseases of our age; in astrophysics it could finally lay bare the secrets of the formation of the universe; in the energy field the impact would be even stronger, as it could allow to better control fusion but also to effectively reduce pollution helping to design innovative cost-effective renewable energy plants. Last but not least, it is believed that Exascale (the next generation computing systems, as of the day of writing) is the order of processing power of the *human brain* at neural level, and because of that, an Exascale system could allow the reverse engineering of a human brain, but also – and more interestingly, though – the possibility to emulate it [14].

Current trends, however, suggest that there is the need to explore new paradigms in designing future systems;indeed:

- Moore's Law, if interpreted *incorrectly* as the doubling of performance every 18-24 months, has hit a *power wall*, as indeed clock rates have been essentially the same since the beginning of 2000s.
- Moore's Law, if interpreted *correctly* as the doubling of the number of transistors on a chip every 18-24 months, is still valid. However, it must be stated that it is impossible to reach Exas-

cale just by doing more of the same but bigger and faster. Indeed, current technology cannot be used to build an Exascale system, as it would likely cost more than 100 billion dollars, and require its own dedicated power plant and over 1 billion dollars per year to be powered [191].

- The attempt to hide the ever increasing memory latency wall by designing larger and more complex cache hierarchies has definitely hit its limit in terms of effectiveness on real applications.
- New parallelization strategies are needed. It is increasingly complex to extract parallelism from sequentially designed programs automatically, but also the distribution of the load onto an enormous number of Processing Elements (PEs) requires a radically different approach.
- The traditional single-domain research activities where hardware and software are explored in an isolated way cannot anymore sustain the growing demand of efficient solutions.

These and other claims make up some of the relevant challenges facing the designers of next generation high performance computing platforms, a matter to which is dedicated the next Section.

1.2 The Challenges of High Performance Computing

While designing a new system that can be competitive with a modern HPC requires a non negligible effort, managing to make a performance leap of orders of magnitude is an infinitely more complex task. There are in fact some important challenges within the HPC field that must be addressed in order to be able to attain such an accomplishment [40]. The focus of this section is to clearly define what they are, and how they impact the design of the next-generation systems.

1.2.1 The Energy and Power Challenge

Power consumption is the most compelling concern, as it is critical to improve overall power efficiency by 2 orders of magnitude in future computing systems. This is because one of the main cost of operating an HPC system, *i.e.* of the operating expenditure (OpEx),

is precisely power consumption. Indeed, assuming a linear scaling of today's best of breed system in terms of performance, the power requirements for an equivalent Exascale system would still be of the order of gigawatts, with an energy cost of more than 2 billion dollars per year. Therefore new serious research challenges arise to achieve a better power efficiency, and it is believed that this will be the area in which significant improvement will be the most difficult to achieve.

Additionally, the majority of the power consumed by supercomputers today is not used to perform computations, but is used to move data around the system [183]. In fact:

- Power consumption increases proportionally to the bit-rate, so as we move to ultrahigh-bandwidth links, it becomes a dominant factor;
- Power consumption is highly distance-dependent, as it grows quadratically with the wire length.

Therefore the emerging constraints on energy consumption will effectively influence the way of designing an HPC system, for example leading to an increase in the usage of optical technologies to perform data movements, also adding to the goals of algorithm design the power constraints as well as an efficient reduction of data movements.

Designing an HPC system with lower power requirement leads to various advantages, first of all the scale down of the cooling system size which in turn involves in cutting the overall costs of the HPC system. Even if there have been substantial improvements in energy efficiency during the last years, HPC continues to be criticized for its extraordinarily high energy demand, leaving a strong need for an accelerated progress.

The U.S. Department of Energy has set the goal of 20MW as the limit of power consumption for an Exascale system to keep its operational costs in a feasible range, whereas modern data centers typically provide that amount of power. However, the most efficient large-scale HPC system, the german *L-CSC* [3], makes us understand how distant the actual technology is from the desired goal, since it is capable of achieving only 5 GFLOPS/W. An Exascale systems would need an improvement of 10× with respect to the L-CSC's power efficiency in order to stay under the limit of 20MW.

1.2.2 The Memory Challenge

The second major challenge is related to storing information, and is due to the lack of currently available technology to retain data at high enough capacities, but also to access it at high enough rates and low enough latencies, still remaining within an acceptable power demand [105, 109] (see Figure 1.1).

Memory capacity using traditional Dynamic RAM (DRAM) technology turns out to be a matter of costs. Current trends show that although the number of cores per processor is increasing, the amount of memory ratio with respect to the available computational capacity is decreasing. This is essentially due to the fact that the cost of memory has not been decreasing as rapidly as the cost of floating point performance, simply because the rate of increase of memory density has never been as rapid as that of Moore's law for the number of transistors on a processor. Even though progresses have been made in the past, as the memory cost has decreased by a factor of over 10^{10} in less than sixty years, current costs are still prohibitive when a very large amount of it is needed.

Memory bandwidth is instead a structural issue rather than a cost issue. In fact, while for processors the demand has ever been for more rapid instruction execution, memory evolution has been guided by the demand of an increase in density to maximize the amount of data available to the processors, resulting in the employment of production technologies that allowed to build large capacity memories, for which, however, the latency was relatively high. This in turn resulted in an ever growing gap between the number of instructions a processor is able to execute and the number of memory transfers that can be done within the same amount of time.

Processor designers addressed this issue by designing hierarchical memories to mask the memory latency. Modern processors are in fact equipped with different levels of memory caches that can store data from DRAM so that future requests for that data are readily available. Cache memories located on-chip are typically built out of Static RAMs (SRAMs), a type of memory built using transistors only. While SRAMs feature vastly lower latencies with respect to DRAMs, they have relatively low data densities and are also more susceptible to errors, so that designers are also required to design them with error correction logic. This translates into overall higher costs, the main reason why they are small-sized and still need to be backed by

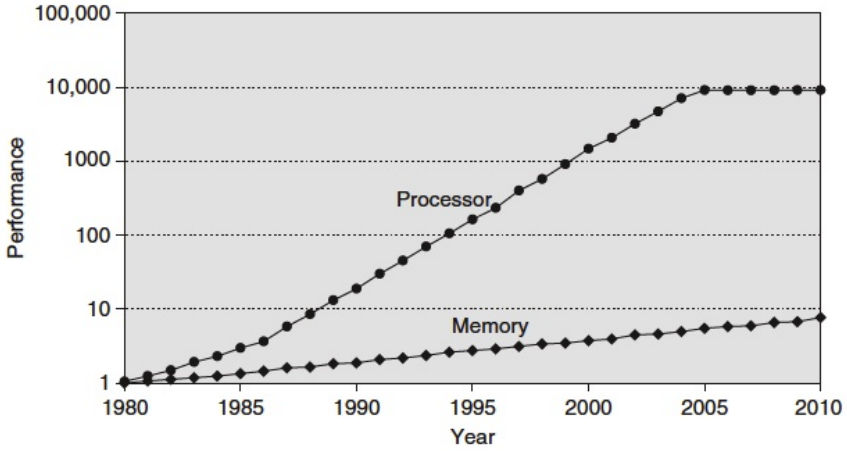


Figure 1.1: *An illustration of the von Neumann Bottleneck. The graph refers to the evolution of canonical CPUs. Performance is measured as the reciprocal of latency. Source: [105]*

traditional DRAMs. Due to the hierarchical nature of memory, it is preferable for a processor to hit a required data block in cache, as it can be accessed more rapidly. Intuitively, when the cache miss ratio is high, the positive impact of the hierarchical memory system can be completely void. Most frequently, high performance programs are indeed coded with a specific architecture in mind, in order to properly exploit both memory bandwidth,latency and cache hierarchies.

To summarize, the memory challenge must be addressed in two different but nevertheless complementary ways:

- providing as much capacity at each level of the hierarchy, but with an acceptable request in terms of cost;
- providing the most effective methods for moving data among the levels as dictated by the needs of the various applications. This is crucial also because memory latency heavily impacts parallel cores performance, essentially due to the inherent need of synchronizations.

1.2.3 The Concurrency and Scalability Challenge

The end of the increase in single compute node performance by increasing instruction level parallelism and higher clock rates has

left explicit parallelism as the only mechanism to increase the overall performance of a system. Mathematical models, numerical methods, and software implementations will all need new conceptual and programming paradigms to make effective use of extreme levels of concurrency. With clock rates flat at several gigahertz, systems will require more than one billion concurrent operations to achieve Exascale levels of performance and most of this increase in concurrency will be within the single compute node.

Concurrency can be measured in three ways:

- The total number of operations that are instantiated in each cycle to run the applications.
- The minimum number of threads that run concurrently to provide enough instructions to generate the desired operation-level concurrency.
- The overall thread-level concurrency that is needed to allow some percentage of threads to stall while performing high-latency operations, and still keep the desired dynamic thread concurrency.

A clear medium-term priority is the definition and implementation of algorithms that are scalable at very large levels of parallelism and that remain sufficiently fast varying latency and bandwidth availability; scalability should be modeled and analyzed mathematically, using abstractions that represent key architectural features.

The increased levels of concurrency in a system greatly increases the number of times that different kinds of independent activity must come together at some sort of synchronization point, increasing the potential for races, metastable states, and other difficult to detect timing problems. It will be necessary to maintain something like a billion threads of control, subdivided into a millions of processors cores to achieve an exaflop. A directly related problem will be the need to make sure that the required data is readily accessible to the computational units. Thus the data must be staged appropriately and the locality of the data must be maintained. Performance scalability of computing systems has been and will continue to be increasingly constrained by both the power required and speed available to enable data communications between memory and processor, but also by the phenomenon known as *dark silicon* [77], caused

by the failure of *Dennard scaling* [71], *i.e.* transistor scaling and voltage scaling are no longer in line with each other. The mere increase of the amount of cores cannot be carried out without exceeding in power density, which in turn can result in the impossibility to keep the chip temperature in the safe operating range. This limitation forces us to systematically power up only a fraction of the entire die, causing large idle or heavily underclocked portions of silicon area, hence the term dark silicon. This phenomenon inevitably restricts the amount of cores a chip can accommodate. These considerations are already giving rise to richer parallelism paradigms, as pure many-core/thread level parallelism are being integrated – at the cluster level – by forms of process-level parallelism, an example of which is *MapReduce* [70].

1.2.4 The Resiliency Challenge

Resiliency is the property of a system to continue effective operations even in the presence of faults either in hardware or software. The vast majority of today's applications assume that the system will always operate correctly. However, an HPC system must be able to use so many components that it is unlikely that the whole system will ever be operating normally, as it is obvious that, the larger the system, the shorter is the mean time between failures (MTBF). The common approach for resilience, which relies on automatic or application level checkpoint and restart, is not suitable for very large systems, as the time for checkpointing and restarting could even exceed the mean time to failure (MTTF), resulting into an irreversible deterioration of the integrity of the system. Also, the problem intensifies when considering that there is the need of handling the lack of resilience of not only computation, but also communication and storage [106].

1.2.5 The Software Challenge

While large scale parallel processors have greatly increased the performance potential for HPC, they have also introduced substantial new software development problems. There are basically two schools of thought regarding the issue of properly adapting software development to the context of HPC. In the first case, the belief is that it is feasible to extract parallelism opportunities from current software, as well as enhance the available paradigms to be able

to deal with the enormous amount of concurrency needed. In the second case, the belief is that a radical rethink is required, and that new methods, algorithms, and tools are needed to enable the performance leap.
The reality is however that both philosophies *must* coexist, and that the actual need is to figure out how to integrate and support existing computation paradigms while enabling new revolutionary paradigms.

At the same time, it is also crucial to provide software developers with the right skills, since up to now there is a serious lack of parallel programming skills across all the degrees of experience, from entry level to very high end. An effort must be also made to raise awareness among HPC users, scientists in the first place, to understand the software challenges and train them to deal with the ever increasing complexity of the systems [106].

1.2.6 Trend Analysis in High Performance Computing

Current technologies cannot deliver increasing processing power on the assumption of Moore's law about number of transistors: such a high number with such a high power density generates too much heating that cannot be dissipated in the limited space of a regular die.

Modern datacenters require large investments in electricity supplies not only to supply electronic equipments but to cool them down, too. Additionally, electrical power drawn by the datacenter is hitting limits imposed by utilities companies in most places, as well. As power efficiency not only reduces costs in electronics but allows bigger savings in conditioning systems, research is focusing on new techniques to reduce the energy consumed by those systems [6].

The most relevant trend harnessing this problem is *heterogeneous computing*. Heterogeneous computing refers to systems that use more than one kind of processing unit. These are systems that gain performance not just by assembling more components of the same type, but by adding customized processing units, usually incorporating specialized processing capabilities to handle particular tasks (and not other). As these components are suited for a specific set of tasks only, a lower number of transistors are usually required in order to process them. Specialized components usually work at lower frequency, too, reducing the overall power consumption of the system – mathematical co-processors where introduced in the late 80's for

similar reasons. This was the first example of heterogenous system. A more relevant and modern example in this direction is the introduction of Graphic Processing Units (GPUs), that were initially used to accelerate the compute-intensive work of texture mapping and polygon rendering. Afterwards, units were added to accelerate geometric calculations such as the rotation and translation of vertices into different coordinate systems.

This is due to the nature of the computation a Central Processing Unit (CPU) is built and optimized for (see Figure 1.3): *irregular computation*. Irregular computation – in this context – means that instruction flows are hardly (if not at all) predictable at compile time, and even when they are, the data access pattern might not be regular at all. Since CPUs' internal structure has a limited amount of logic dedicated to actual computation, only a relatively lower number of numeric operations can be performed at a time. On the other hand, GPUs, due to their simpler and parallel internal structure (see Figure 1.3), are better suited to scientific computation as they provide multiple identical components that can simultaneously execute the same instruction. Even if GPUs' consumption are very high, given an highly (data) parallel workload, they are capable of delivering much more FLOPS per Watt due to the intrinsic parallelism of their architecture and the amount of logic actually designed to computation.

Another reason for CPUs vs GPUs power efficiency is due to the abstraction layer that implements its software programmability. For this reason, GPUs have been extended in the last decade to support generic computation and are the current heterogeneous component of election (at least in the high performance computing sector).

Another important direction in heterogeneity is the introduction of physics chips: they offload physics calculations from the CPU, and are performed on dedicated hardware circuit (for example PhysX [11], is a proprietary realtime physics engine middleware SDK, born from an hardware solution by Ageia, that called it Physics Processing Unit (PPU)).

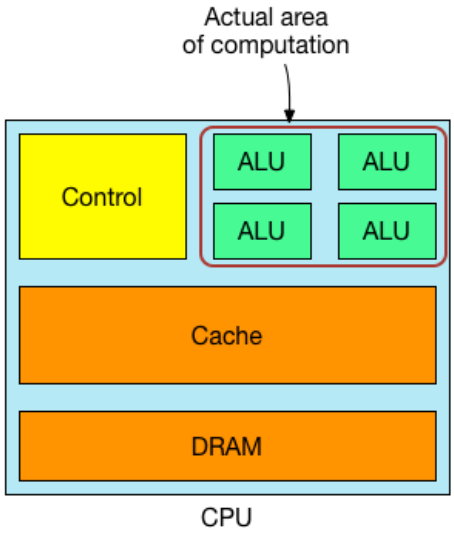


Figure 1.2: *A representation of a CPU – the simplistic model underlines how only a relatively small portion of the device is dedicated to actual computation.*

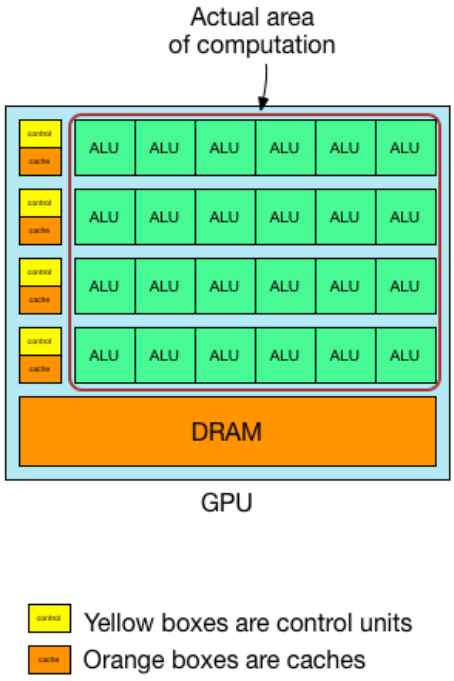


Figure 1.3: *A representation of a GPU – the simplistic model underlines how a relatively large portion of the device is dedicated to actual computation.*

Phi cores [5] are based on the same idea of the multicore architecture, but relying on more, less complex micro processors. The basic idea that led to the design of such component is that these cores can retain many of the existing programming models (and some tools) that most developers are familiar with.

Trends show that we need to find a different approach that reduces power consumption, while increasing power efficiency and parallelization. Elaborating on these and other trends, we look forward to a component that transcends this abstraction layer and uses all the power it drains to make effective computation.

Among the other heterogeneous computing technologies, FPGAs show an interesting set of features that suggest that these devices will become a relevant computing fabric in the near future. FPGAs, differently from CPUs or GPUs, don't feature specific low level circuitries, to be programmed using a given language; instead, they feature high level hardware blocks implementing specific functions like: logic functions emulators, digital signal processing blocks (featuring plenty of multiply and accumulate – *MAC* – stages), and basic so called block RAMs. Depending on how these blocks are connected, computation takes place accordingly to designer's will. This technology allows the designer to implement very customized logic circuits in order to accelerate the "hot portion" of target applications (or entire applications, depending on the context). This customization process reduces, on average, the number of transistors required to implement a specific functionality (compare this against the complexity of a deeply pipelined, out of order, superscalar processor, and its billions of transistors) – this way, it is possible to achieve increased power efficiency figures.

However, FPGAs must be properly configured in order to obtain power efficient processing units out of them. This process is very complex (with respect to software designs only) as it involves both careful analysis of the target application and multiple hardware design steps. Additionally, it must be repeated for every problem at hand, resulting in a very time consuming process.

However, the design of custom architectures tailored around a specific algorithm will result in huge energy savings, incrementing the power efficiency of the system. Not all the problems can take advantage of this approach, but many can be efficiently implemented.

Programmability is an issue as FPGAs are programmed in a very different manner than CPUs and GPUs. Current research – both in-

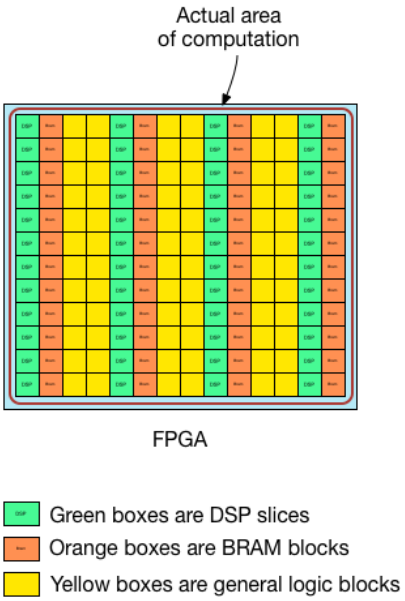


Figure 1.4: *A representation of an FPGA – the simplistic model underlines how most of the device's area is dedicated to actual computation.*

dustrial and academic – is focusing on improving the experience of software developers as they should concentrate on software algorithms only, leaving a sophisticated toolchain the burden to implement it as dedicated set of circuits (for example, Xilinx [213] with SDAccel SDK [13]).

1.3 Heterogeneous Systems

The majority of existing supercomputers generally achieve only a fraction of their peak performance on certain portions of some application tasks. This is because different subtasks of an application can have very different computational requirements that result in different needs for processing capabilities. An homogeneous architecture cannot satisfy all the computational requirements in certain applications equally well.

Thus, the construction of an heterogeneous computing environment (like that in Figure 1.5) is more appropriate. Employing an heterogeneous system can be the solution to properly meet all the presented challenges, as it offers the opportunity to increase the compu-

tational performance keeping low the energy requirements. Heterogeneous computing [185] refers to systems that use more than one kind of PEs, each of which is particularly efficient within a specific application domain. These PEs communicate through a system of high-performance interconnections. To take advantage of such a system, a given task is decomposed into subtasks, where each subtask is computationally homogeneous, and assigned to the PE whose characteristics are the most appropriate to its execution. One or more PEs, being canonical CPU, are in charge of managing the offloading to the other PEs, as well as the execution of general purpose components of the computation such as operating system services.

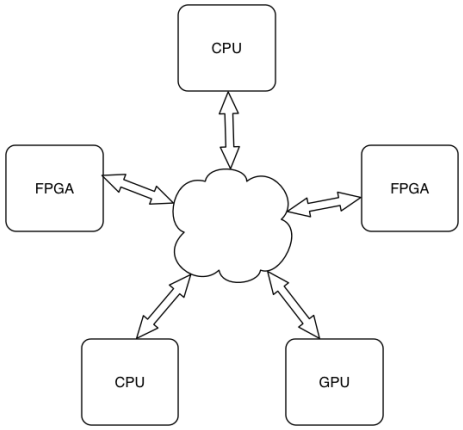


Figure 1.5: *A simple scheme of an heterogeneous system.*

The rationale beyond the employment of an heterogeneous system is that CPUs are designed to handle complex control flows, but their general purpose nature makes them unfit to retain a high and cost effective throughput whit respect to other available solutions. CPUs are then coupled with other coprocessors, namely General Purpose Graphic Processing Units (GPGPUs) and FPGAs, both of which have specific characteristics that make them suitable to perform certain kinds of computation. GPGPU, being Single Instruction Multiple Data (SIMD) processors, perform very well on highly data parallel tasks. They have a massively parallel hardware architecture, are capable of achieving high floating point performance and have large off-chip memory bandwidth, however it is usually very difficult to make GPGPUs work at their full capacity and use all the available bandwidth. Also, for high end chips the power

demand can be huge, even though they are still capable of delivering high power efficiency – at least with respect to conventional CPUs, which justifies their employment as co-processors in heterogeneous systems. FPGAs offer very high I/O bandwidth and fine-grained, custom and flexible parallelism. They are mainly composed of three building blocks [108] (see Fiure 1.6): the Configurable Logic Block (CLB) is the main component, it can implement one or more function generators using Look-Up Tables (LUTs) which in turn implement an arbitrary logic function, storing the result of the function for every possible combination of the input. The Input-Output Blocks (IOBs) are in charge of connecting the signals of the internal logic to an output pin of the FPGA package. The interconnection resources allow the connection of CLBs and IOBs. An FPGA can have additional resources embedded on the die, such as Random Access Memory (RAM) cells (also called Block RAM (BRAM)) that can be used to store data on-chip during the computation, Digital Signal Processors (DSPs) and other specific processors.

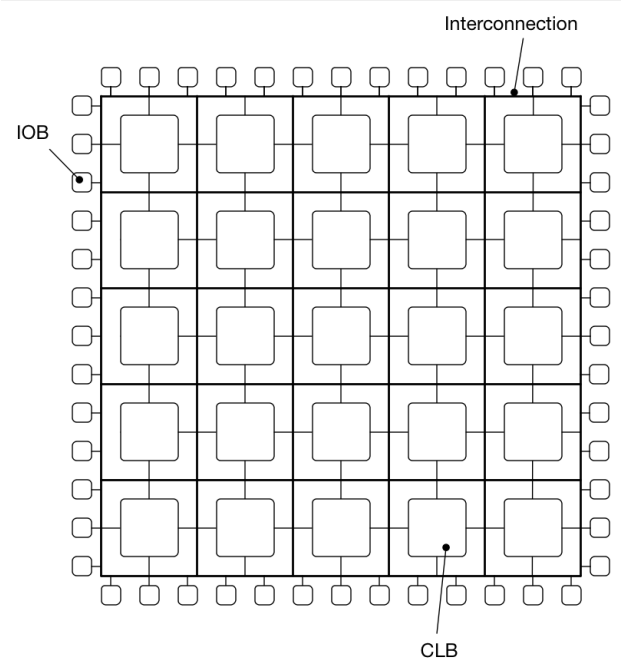


Figure 1.6: *The general architecture of an FPGA.*

The structure of an FPGA enables tasks-tailored logic to be cre-

ated on-the-fly, that, considering the ever-increasing computational needs coupled with the frequency/power wall, is the perfect solution to have both performance and low power consumption. Indeed, the employment of custom logic, shaped on the specific type of computation, allows to have, within the entire fabric of an FPGA, only the part demanded to implement the circuit to be powered on. Therefore, an efficiently designed custom logic can lead to both sustained performance and low power consumption, as previously stated, thus high power efficiency. High Performance Reconfigurable Computers (HPRCs) based on conventional CPUs and FPGAs as coprocessors have indeed been gaining the attention of the HPC community in the past few years. In these systems, the main application executes on the CPUs, while the FPGAs handle kernels that have a long execution time and are suitable to hardware implementations. Such kernels are typically data-parallel overlapped computations that can be efficiently implemented as fine-grained architectures. Optimization techniques such as overlapping data transfers between the CPUs and FPGAs with computations are useful for data-intensive, memory bound applications.

However, there is an underlying complexity in heterogeneous systems that simply cannot be handled with modern software solutions, as different architectures must be programmed in different ways. Therefore, there is the need to provide automatic solutions capable to hide the complexity of these systems, as well as promoting the adoption of techniques that bring together software and hardware design, the so called *co-design*, which is believed to be a promising solution to make Exascale computing a reality [32].

1.3.1 Metrics

Different HPC systems have in general really different architectures, employ a variety of computing devices and handle data movements with different approaches. Hence, there is the need to define some standard metrics that can be used as fair terms of comparison. A fair comparison of computing systems is essential, particularly so within the HPC field; for this, the research community standardized a number of significant metrics (and depending on the system, also well-defined measurements procedures [4])

- Throughput, measured in FLOPS, Floating Point Operations per Second

- Energy-Delay product, measured in J·s
- Total Aggregated Bandwidth, measured in bit/s,
- Total Aggregated Absorbed Power, measured in watts (W), and
- Power Efficiency, measured in FLOPS/W.

In this thesis we design prototypal systems while focusing on the optimization of our designs in terms of either energy latency, throughput, and/or power efficiency.

1.4 Hardware Acceleration

In this section I describe what hardware acceleration is and why it is employed to achieve higher power efficiency than today's high performance computing systems.

1.4.1 What is Hardware Acceleration

Hardware acceleration is a technique that consists of implementing some, or all, parts of an algorithm via dedicated hardware circuits (as for example in Figure 1.7). Said circuits produce the same results as their software counterparts [20, 58, 147, 164]. Traditionally, the hardware designer was in charge to creating the circuits by hand. The designer traditionally required a complete understanding of hardware components and how they could be connected in order to achieve the corresponding algorithmic operation. The entire workflow is very time consuming, involved and error prone but nonetheless required when the goal is to achieve the best performance available. This workflow will make extensive use of Hardware Description Language (HDL), which are difficult to understand and manage for most software designers who usually are the ones in charge of coding algorithms. To make a comparison between hardware and software development, HDL based development resembles the use of Assembly to optimize custom routines in C/C++ development. Since HDLs were developed to describe hardware circuits, they are characterized by a low level of abstraction. Thus, hardware designers must take into account every single detail such as signals, state machines and their behavior over time. Also, debugging at this level is very complex and an hardware and electronic knowledge is required to understand waveforms, timing constraints

and their impact in the final design.

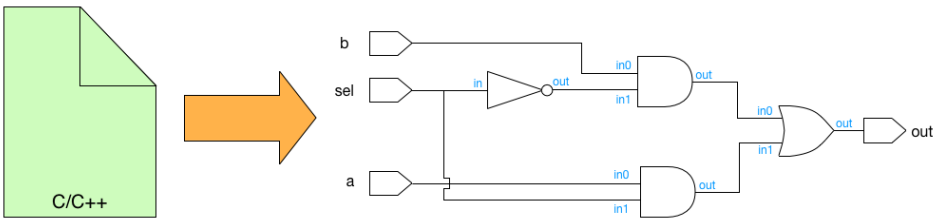


Figure 1.7: *Possible hardware implementation of IF statement*

1.4.2 Why to employ Hardware Acceleration

The implementation of a dedicated hardware component has the major benefit of speeding up portions of an application. In fact, it is usually true that there is no CPU program that can run as fast as a dedicated circuit given that the latter is comparable in terms of technology and frequency to the former. This is due to the overhead needed to maintain the CPUs as a general processor as possible (i.e. in order to compute anything the software developer can think of).

This is also the reason why GPUs were introduced: a dedicated hardware capable of running specialized instructions to compute graphics-like processing (i.e. data parallel codes) very fast. This kind of device features a lot of dedicated circuitery to do a specific task, such as transform geometric primitives or triangle setup/-clipping; nevertheless, a lot of small and simple processing units run in parallel, achieve better performance than a CPU in graphics computation. Additionally, even if the GPUs were invented to do graphics computation, in recent years it is becoming more and more common to exploit their intrinsically parallel architecture to achieve better performance on specific workloads, like scientific computation. These devices are more difficult to program than CPUs (mainly due to the heterogeneous nature of the resulting system) but can be programmed in a similar fashion. However, few drawbacks affect GPUs, in order to maintain the processing as general as possible for computation. For example, although GPUs usually feature high throughput and very high internal memory bandwidth, it is usually very difficult to make GPUs work at their full capacity and rarely saturate the internal bandwidth.

As programming GPUs is a very complex task, major vendors put a lot of effort into introducing a set of Application Programming Interfaces (APIs) and libraries to make the process easier. Notable examples are Nvidia Cuda [2] and AMD Mantle [8] frameworks. Moreover they show relatively low power efficiency with regards to FPGA [53, 63, 197] on most workloads, for the aforementioned reasons.

We pay overhead when we have a lot of data dependent behavior inside the application. If the algorithm is static and every implementation detail can be known at compile time (apart from the true values of the data needed to process) then we can create a very small circuit that operates very fast multiple times, requiring less time and far less power.

To summarize, GPUs can be considered suboptimal for high consumption and relatively low power efficiency (at least, compared to FPGAs).

This is where FPGAs play an important role. As we can tailor the processing system around the application, by stripping away all the intermediate steps, we achieve higher power efficiency. Unfortunately, the development of even a small component is a very complex process. In recent years, in fact, a lot of effort was put into automating the creation of such systems by means of High Level Synthesis (HLS). HLS tools synthesize circuits from languages such as C or C++ instead of the less handy VHDL or Verilog, enormously speeding up the development of hardware based systems.

1.4.3 High Level Synthesis

While HLS tools have been heavily studied in the past, only in the recent years we have seen effective industrial tools available in the market [88]. Current research is focusing on efficiently converting numeric or image processing algorithms written in behavioral languages directly into hardware implementations in order to achieve better performance and lower consumption while lifting the layer of abstraction in order to gain in designer programmability [26, 27]. This has been possible in the recent years because High Level Synthesis (HLS) tools have become powerful and flexible enough to allow relatively easy and fast synthesis of hardware circuits. Previously, hardware development required plenty of specific knowledge in order to develop a fully working accelerator. High Level Syn-

thesis (HLS) tools impose less requirements on designers and dramatically speeds up the development of a working system. However, without proper care, this comes at the cost of introducing large overheads and slow-downs compared to manually designed implementations. This is due to the lack of knowledge that High Level Synthesis (HLS) tools have in order to do optimizations on the resulting components. In order to cope with these limitations, High Level Synthesis (HLS) tools have special directives that can be used to optimize the resulting components, with the only downside that these directives need to be specified by the designer and are not derived automatically.

1.4.4 Optimized High Level Synthesis

HLS tools usually feature lots of directives employed by designers to signal the synthesizer how to generate different components [64, 88], each with its own specific performance profile. Some of them are useful to increment the throughput, others to minimize the area and others again are explicitly used for lowering the power consumption. For example, the *dataflow* directive can be used to parallelize function calls and/or nested loops creating different blocks of circuits inside a single core, each capable to run concurrently with each other. This directive also preserves the data dependences of the input code to maintain the correctness of the output. Another useful directive is the *pipeline* directive. This directive tells the HLS tools to use more resources in order to create a pipeline inside the core, or in case this directive is used with the *dataflow* directive, to create a pipelined block inside the core. Directives such as *array map*, *array reshape* or *array partition* serve the purpose to optimize the number of BRAMs used inside the FPGA. Finally, the *unroll* directive can partially or completely unroll a loop in order to run in parallel all its iterations of a loop body (see [88]).

1.4.5 Input languages to HLS tools

Current hardware circuits can be generated in very different ways.

First of all, we can generate a Register-Transfer Level (RTL) description of the circuits from manually derived VHDL or Verilog, each describing the hardware behavior. This is the standard, inefficient workflow in hardware design.

As previously stated in 1.4.2 and 1.4.4, HLS tools are getting more and more powerful, closing the gap between automatic and manual implementation; plus, they allow the creation of RTL from high level language such as C/C++, or with the newer OpenCL C compilers from both Altera and Xilinx.

The reason why HLS vendors choose C/C++ is a three fold argument:

- The vast majority of the legacy code for numeric computation is written in C/C++
- Designers are already productive and familiar with imperative/procedural languages such as C/C++
- Designers can rapidly explore the impact of standard directives (i.e design modes) to find better trade offs between latency, area used, power consumption and throughput

While these are industrial considerations we cannot overlook, there are other reasons to choose C/C++, namely:

- most syntax analyzers and compilers are written for C/C++ so its easy to get robust tools to further enhance code deriving from them
- It's easy to simply port algorithms from a platform to another and to HW, too, as C is well defined and standardized
- Support a familiar "hardware level of abstraction", providing a link between high-level source code and low-level implementation [96]

On the other hand, other languages can be better as they can leverage different, more hardware friendly formal semantics to produce better parallelizable codes.
Those features are, among others:

- No aliasing (i.e. Fortran)
- All parameter passing is done by value (we solve from language itself some synchronization issues, i.e. Haskell, but we do not resolve communication issues)

- Passing arguments by value will waste memory very quickly and so we need to rethink the algorithm in a more efficient way
- Atomic guards (BSV)
- Explicit memory hierarchy stores (partially, OpenCL)

Note that other languages can also use other means to get parallel/optimized computation: for example, in Haskell you get for free fast lightweight threads, parallel sparks and futures, software transactional memory, core affinity control and so on. However, such features mostly cannot be ported to HDL (even if there are projects like [26,27,76,120,121,127,128,194] that aim for it).

However, since the leading industry focuses on subsets of C-like syntax languages, for the rest of the thesis I will consider HLS tools targeting C/C++.

1.5 Thesis Contributions and Outline

Within this context, the work proposed in this thesis embraces the principles of heterogeneous computing to make a little step towards the achievement of the Exascale milestone by following three different but related research lines.

First of all, we develop an accelerator-rich platform where the focus is on the coordination of multiple custom and software processors, via a novel Domain Space Exploration (DSE) phase. Specifically, the work elaborates on two relevant aspects: the effectiveness of Partial Reconfiguration (PR) to attain improved energy delay and throughput metrics, and the effectiveness of the heuristics chosen to realize the DSE step, which feature both low complexity and good exploration times. To this matter, we devote Chapter 2. We also extended the scope of this work by assuming that multiple computing elements are in place, and an adequate communication architecture is required to coordinate those accelerators. This is discussed in Chapter 3.

Secondly, we focus on amply data-parallel codes, and develop a novel HLS approach to using PM as a means to explicitly extract and isolate data and computation from affine codes in order to efficiently divide the workload among an arbitrary number of nodes, in the light of the current and foreseeable trend of adoption of reconfigurable hardware in the datacenter; towards energy proportional

computing, we improve the current state of art in single core acceleration, as our methodology obtains near-linear speedup with the area at disposition to accelerate the given workload. To this subject, we devote Chapter 4 to introduce the background notions related to Polyhedral Analysis (PA), and Chapter 5 to delineate the methodology and the results.

Lastly, we focus on a specific, and more restricted class of data parallel codes, namely Iterative Stencil Loop (ISL), as they play a crucial role in a variety of different fields of application. The computationally intensive nature of those algorithms created the need for solutions to efficiently implement them in order to save both execution time and energy. We introduce the notion of Streaming Stencil Time-step (SST), a *streaming-based architecture* capable of achieving both low resource usage and efficient data reuse thanks to a demonstrably *optimum* data buffering strategy; and we introduce a technique, called SSTs *queuing*, capable to deliver a *quasi-linear* execution time speedup with constant bandwidth. To these extensive methodologies, we devote Chapter 6 and Chapter 7.

Sources This thesis refers to the following materials:

- **SMASH: A Heuristic Methodology for Designing Partially Reconfigurable MPSoCs**, by R. CATTANEO, C. Pilato, G. Durelli, M. D. Santambrogio, and D. Sciuto. Appeared in the proceedings of IEEE International Symposium on Rapid System Prototyping (RSP), 2013.
- **K-Ways Partitioning of Polyhedral Process Networks: a Multi-Level Approach**, by R. CATTANEO, M. Moradmand, D. Sciuto, M. D. Santambrogio. Appeared in the proceedings of Reconfigurable Architecture Workshop (RAW), 2015
- **Explicitly Isolating Data and Computation in High Level Synthesis: the Role of Polyhedral Framework**, by R. CATTANEO, G. Pallotta, D.Sciuto, M. D. Santambrogio. Appeared in the proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2015
- **On how to Accelerate Iterative Stencil Loops: A Scalable Streaming-based Approach**, by R. CATTANEO, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio. Transactions on Architecture and Code Optimization (TACO).

CHAPTER 2

Design Space Exploration of Partially Reconfigurable Hardware Accelerators

In this Chapter we analyze the impact of Partial Reconfiguration (PR) on the overall performance of the computing system. More specifically, we define an architectural template that we iteratively customize and objectively evaluate as part of a novel Domain Space Exploration (DSE) phase. As part of this design space exploration, PR is modeled and introduced as a possible degree of freedom. After describing the state of art and the overall methodology, we show how the introduction of PR is an effective technique to improve the overall performance of the resulting reconfigurable hardware based accelerators.

2.1 Introduction

Nowadays, the design of efficient embedded systems relies on heterogeneous Multi Processor System on Chips (MPSoCs) [142] that combine general purpose processors with dedicated hardware accelerators. Indeed, efficient tools (e.g., Xilinx Vivado HLS, Synopsys

C Compiler, Cadence C-to-Silicon, Calypto CatapultC) are becoming very popular for the automatic generation of hardware implementations from the corresponding behavioral specifications, also allowing the possibility to explore different implementations (e.g., trade-offs between performance and requirements of resources). As a result, high-performance and low-power architectures can be obtained with the hardware acceleration of different parts of the application [74], even if their design still requires high expertise.

In particular, Field Programmable Gate Arrays (FPGAs) are very attractive solutions that allow implementing large parts of the application in hardware at low cost. Moreover, exploiting *Partial Dynamic Reconfiguration* [178] (PDR) offers the possibility of reusing some part of the logic across different tasks, despite of an overhead in the execution time required to reconfigure the corresponding logic cells. For this reason, this technique introduces several challenges that have to be properly taken into account during the design of such systems. In particular, one of the main issues in designing heterogeneous systems, especially when PDR is taken in account, is the customization of the architecture [174] in terms of hardware accelerators (either static or reconfigurable) that are usually defined in advance, potentially leading to sub-optimal solutions. Then, the designer has to determine which tasks have to be hardware accelerated and the level of reconfiguration for each of them (if any), also with respect to the number of available resources. Moreover, since a task reconfiguration requires to load the new configuration bitstream for the corresponding region, this can introduce a penalty in the execution time if it is not properly taken into account in the design of the application [29]. Finally, the effects of the data transfers between the tasks are crucial aspects [85] and thus the impact of the interconnection infrastructure (e.g., bus, NoC, FIFO) has to be necessarily taken into account. In conclusion, novel and efficient methodologies are definitely required to take into account the reconfiguration aspects for the early stages of the design process.

In this chapter, we propose *SMASH* (Simultaneous Mapping and Scheduling with Heuristics), a design methodology that aims at addressing the limitations cited above. It combines different heuristics for both customizing the architecture and implementing the application to generate reconfigurable systems tailored for the input partitioned specification. Indeed, it determines which implementation has to be adopted for each task and the level of reconfiguration of

the corresponding hardware modules, also potentially taking into account different topologies that can adopted for interconnecting the processing elements. In fact, SMASH includes an exploration phase that is able to determine the proper mapping and scheduling for the different tasks of the application, determining the tasks' implementations and the processing elements where they have to be executed. During this exploration, it also determines which hardware modules have to be included into the reconfigurable logic (either to be used as static or reconfigurable regions) and the resulting reconfigurations are taken into account during the evaluation of the solution, along with the required communications. Moreover, taking into account the resulting resource requirement of the different regions during the exploration allows to limit the generation of unfeasible solutions. Simulations by means of virtual platforms have been adopted to validate the proposed approach.

The rest of the chapter continues as follows. Section 2.2 overviews existing approaches that aim at addressing similar problems, highlighting the contributions of the proposed solution whose overall organization is presented in Section 2.3. Then, Section 2.4 presents the heuristics at the basis of this work and the solution evaluation method, respectively. In Section 2.5 presents the experimental evaluation of the proposed approach, while Section 2.6 concludes the chapter and outlines the future directions of work.

2.2 State of Art

The synthesis of heterogeneous MPSoCs usually requires an efficient exploration of the design space. Daedalus [195] is an interesting and integrated framework for starting from a sequential application and then generating the corresponding parallel implementation, along with the corresponding system. However, it only focuses on streaming applications and reconfiguration aspects are not taken into account. It is worth noting that these aspects are usually not taken into account for such applications due to synchronization issues in the FIFOs when reconfiguring the blocks. Moreover, single stages of this computation are usually quite simple and, thus, reconfiguration is usually not attractive in this scenario. On the other hand, task-based applications are usually characterized by time-consuming computational blocks interleaved by data transfers. They are usually represented as Direct Acyclic Graphs (DAGs), as

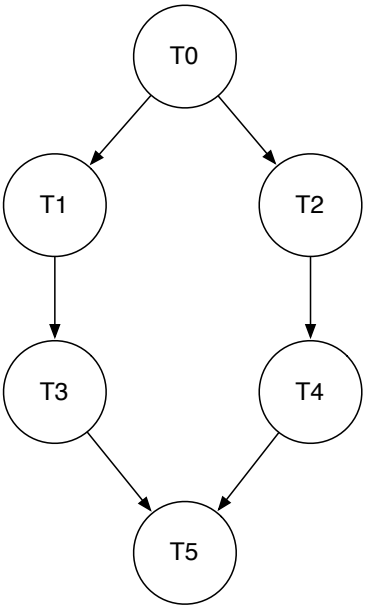


Figure 2.1: *Example of application DAG.*

the example shown in Figure 2.1.

As opposite to creating full-custom architectures, platform-based design [174] is instead a viable solution for reducing the complexity of designing such systems by a a progressive refinement of an architectural template. Based on this idea, different approaches [56, 85, 119] have been proposed for optimizing partitioned applications, especially in the case of hardware acceleration. In particular, [119] uses an approach based on task clustering, while [85] explores different mapping and scheduling alternatives with a constructive approach for limiting unfeasible solutions. However, in both of the cases, PDR is not addressed and thus tasks can be executed in hardware as long as they fit in the available area. On the other hand, [56] proposes a method for mapping and scheduling of reconfigurable systems, but the target architecture (e.g., the number of reconfigurable regions) have to be defined in advance, potentially leading to sub-optimal solutions.

In [100], the authors propose a set of techniques focusing on the partitioning of the code and the generation of the corresponding adaptive system. However, they mainly focus on dynamic aspects and the support for the Operating System (OS), while we are more interested in design-time decisions, in order to have a very lightweight

OS or even a bare-metal synchronization of the application.

On the other hand, it is worth noting that, for creating a feasible implementation of the system, another step is usually required: the definition of the physical constraints within the FPGA to satisfy the resource requirements (e.g., LUTs, BRAMs, DSPs) of the hardware modules and to avoid their overlapping. The authors in [29] propose a methodology for mapping the tasks to processing elements taking into account reconfiguration aspects and also placement issues. However, their assumptions are quite simplistic (e.g., homogeneous resources for the regions) and they are difficult to be applied to recent FPGA devices, where the designer can design very different and two-dimensional reconfigurable regions. In this chapter, we separate the problems: we identify the number of hardware modules while exploring the mapping of tasks with respect to them. Then, we only verify that the total amount of resources required by the regions can be effectively satisfied by the target FPGA. Extending the proposed approach to integrate the verification of the physical constraints is straightforward: a floorplanning algorithm (e.g., the one proposed in [45]) can be integrated in the evaluation of the solution and return to the exploration algorithm if the assignment results in a feasible allocation of the resulting regions or not. However, this is out of the scope of this work and it has been left as a future work.

In conclusion, the main contributions of the proposed approach can be summarized as follows:

- it optimizes the execution of the given task graphs with respect to an architectural template, determining the level of reconfiguration for each of the tasks that are decided to be executed in hardware and the nature of the hardware modules to be introduced in the final platform;
- it defines an exploration framework that can easily accommodate different algorithms, metrics and evaluation methods to design a reconfigurable system;
- it generates the specification of the virtual platform corresponding to the identified solution.

This approach has been validated by means of synthetic applications that are representative of real-life applications. The corresponding solutions have been then evaluated with high-level simulations of

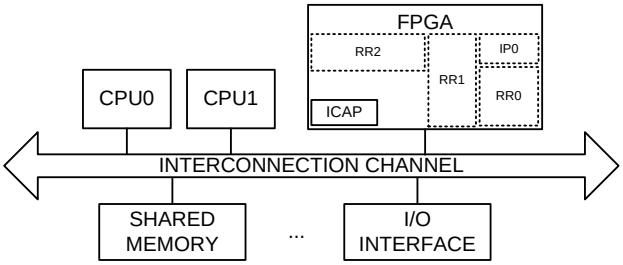


Figure 2.2: *Example of target architecture.*

the generated virtual platforms through Synopsys Platform Architect [192].

2.3 Proposed Methodology

The proposed methodology, namely SMASH, starts from the description of the architectural template to be customized (as the one shown in Figure 2.2) and of one or more partitioned applications to be concurrently executed. Each of these applications can be represented as a DAG (as the one shown in Figure 2.1) and a unique representation can always be obtained by combining them. A list of admissible implementations (e.g., combination of performance and resource requirements) has to be also provided for each task. For software tasks, they can be computed by profiling or estimating its execution. On the other hand, for hardware tasks, it is possible to obtain both execution time and required resources by estimations or by actual synthesis through a HLS tool such as Vivado HLS [15].

SMASH is then applied to this resulting DFG and it is mainly composed of two parts, as shown in Figure 2.3:

1. an exploration of the mapping and scheduling for the input DFG to statically determine which implementation has to be adopted for each of the tasks and where they have to be executed (e.g., processors or reconfigurable logic).

2. a final customization of the architecture, where it is possible to identify static IP cores (i.e., modules with only one task associated with), as well as reconfigurable regions.

As output, it produces a description of the system to be implemented, including the specification of the customized architecture and the

mapping and scheduling of the tasks (including reconfiguration ones) with respect to this generated architectural solution.

In details, the exploration heuristic (further detailed in Section 2.4.2) aims at evaluating different solutions in terms of mapping and scheduling to determine the best implementation for the given DFG with respect to the target architecture enhanced with hardware modules. In particular, when assigning multiple tasks to the same region, the heuristic is able to automatically compute its overall resources requirements and, taking into account the requirements of all the blocks, whether the solution is feasible or not. Then, the solution evaluation determines the reconfiguration tasks that have to be introduced (i.e., when consecutive tasks executing different functions are assigned to the same module) and evaluates the performance of the solution taking into account also the reconfiguration overhead. Note that this term is computed on the basis of the size of the reconfigurable region (i.e., its requirement of resources) as it results from the generated mapping.

The last phase of the methodology is a post-processing step that analyzes the mapping solution and the architectural instance generated after the first phase. In such a situation, each hardware module that has been introduced by the exploration algorithm can have one or more tasks assigned. If the module has only one task assigned, it means that it can be converted into a static IP block, since no reconfiguration is required, thus reducing its area consumption. Otherwise, modules with more than one task assigned are represented as actual reconfigurable regions in the final architecture.

In conclusion, the methodology can produce the description of the resulting system. In particular, we assume that the generated solution performs correctly from the functional point of view and then we are only interested into evaluating non-functional properties of the system (e.g., performance). For this reason, in this work, we generated a virtual platform for the high-level evaluation of the solutions, where each processing element is represented as a Virtual Processing Unit (VPU) and all the VPUs are interconnected as specified by the input architectural template. Then, the initial DAG is mapped onto these VPUs as specified by the generated mapping solution and the reconfiguration tasks are assigned to the VPU that has in charge of performing the actual reconfiguration (e.g., a dedicated processor or one of the available GPPs) to correctly model its execution overhead.

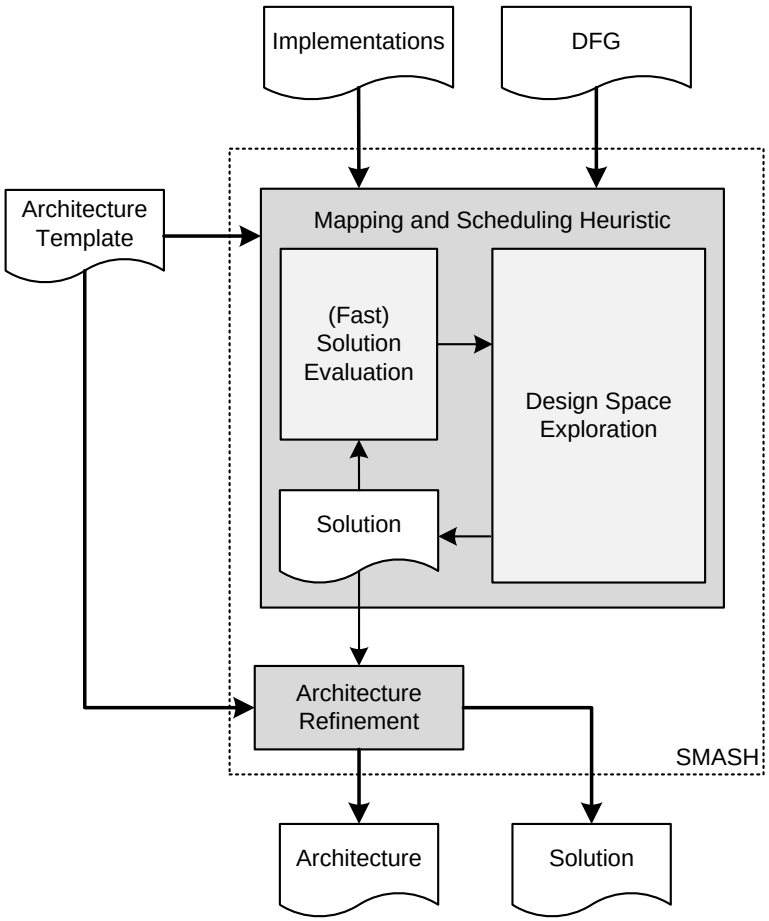


Figure 2.3: *Overview of the proposed methodology.*

2.4 Mapping and Scheduling Exploration

Our mapping and scheduling exploration is based on Ant Colony Optimization (ACO) [66]. This meta-heuristic relies on the abstraction of an agent (i.e., the "ant") stochastically exploring the design space, described as a sequence of choices. At each step, the agent computes all the available choices and ranks them according to a rule which helps the algorithm finding the best solutions, iteration by iteration.

The rule for ranking choices relies on two heuristics: *local* and *global*. The former is a function that assigns a score to a choice given the current state of the exploration, and integrates knowledge about the specific problem. It allows the agent to make an informed (yet

local) decision about the next step of the exploration. The latter is a model for the abstraction of ants' *pheromones*, a substance ants naturally release on the path they traverse and to which are naturally attracted. Being a volatile substance, the intensity of the pheromone trail will eventually disappear at a rate inversely proportional to the number of ants traversing (i.e. reinforcing) it. On the other hand, on a heavily traversed path the deposited pheromone trail will be reinforced, and more ants will be attracted to it. If a path to a target is optimal, ants will reach it in lesser time than other paths; thus, the amount of pheromones on that trail will evaporate relatively slower because it takes less time to traverse it (in other words, it is reinforced more frequently). Since the intensity on that trail is relatively higher, relatively more ants will be attracted to it, releasing themselves more pheromones thus reinforcing again the trail. In a sense, the amount of pheromones on a path globally keep track of how good that path is to reach the goal.

An ACO algorithm implements the exploration in an iterative way by using "generations of ants". A *generation* is composed of N ants performing design space exploration (DSE), all using the same values of the global pheromone matrix. At the end of each generation, only some ants (i.e., a parameter K defined by the user) are considered for global heuristic reinforcement, proportional to the quality of the corresponding solution. After a convergence criteria is met, the best solutions are kept and the others are discarded.

2.4.1 Detailed overview of the algorithm

This algorithm describes the steps required to map an application represented as a DAG onto a reconfigurable architecture composed of different processing elements (i.e., software processors, static IP cores and reconfigurable regions) using the ACO technique with the goal of minimizing the total execution time.

The algorithm evolves a certain number of generations of ants (lines **1**-**22**) until termination criteria is met. In our case, the termination criteria is the number of generations to evolve. Each ant builds a solution as a sequence of *choices* (lines **2**-**22**): in our specific case, they are mapping choices, i.e., 3-tuples of the form $< task, implementation, processing\_element >$ representing how a task should be implemented and where. In this context, an implementation is one of the available ways to execute a task either in

Algorithm 1: Overview of exploration algorithm

```
input : A task graph and a reconfigurable architecture
output: A mapping trace
forall the numGenerations do
    forall the antsPerGeneration do
        readySet ← tasksWithoutPreds ()
        scheduledSet ← ∅
        while readySet ≠ ∅ do
            forall the T_i ∈ readySet do
                p_{T_i} ← localHeuristic (T_i)
            end
            chosenT ← roulette (p_T)
            iSet ← implsOfTask (chosenTask)
            forall the I_i ∈ iSet do
                pSet ← processorsPerImpl (I_i)
                forall the P_i ∈ pSet do
                    p_{I_i,P_i} ← localHeuristic (I_i, P_i)
                end
            end
            chosenI, chosenP ← roulette (p_{I,P})
            choice ← <chosenT, chosenI, chosenP >
            mappingTrace.add (choice)
            readySet ← resolveDependencies ()
        end
        ant.Metrics ← computeMetrics (currentAnt)
        ant.Objective ← computeObjective (ant.Metrics)
        thisGenerationSolutions.add (ant)
    end
    bestAnts.add (selectBest (thisGenerationSolutions))
    updateGlobalPheromones (bestAnts)
end
bestAnt = selectSingleBestAnt (bestAnts)
return bestAnt.trace
```

software or in hardware. An ant computes a complete *trace*, i.e. a list of mapping choices where an explicit scheduling priority is expressed by the ordering of the choices in the list itself.

In order to iteratively build a solution, the ants generate all feasible choices at each step of the algorithm, ranking each of them according to a rule based on two heuristic functions, one for the choice of the task to map (lines **6-8**) and one to choose both the implementation and the processing element to execute this task onto (lines **10-14**). Note that the exploration starts with a minimum number (min_{HW}) of hardware modules to employ into the final architecture. Then, at each step, the choice may reuse the allocated modules or

may instantiate an additional hardware module in order to better exploit hardware resources, given that the area constraint is never violated and the total number of hardware modules is less than a maximum value (max_{HW}) or the number of tasks. After assigning a score to each mapping choice (line **13**), the ant selects one of those choices using a roulette wheel selection scheme (line **14**), where the probability of each choice is proportional to its heuristic value. After a decision for this iteration is made, the ant computes which tasks may be executed afterward (line **17**), and the process continues until no more tasks need to be scheduled (lines **5**-**17**).

At this point, the solution is evaluated according to a reconfiguration-aware scheduler, and an objective function is assigned to it (line **19**). When the last ant of the current generation ends, the best solutions are used to reinforce the global pheromone matrices (line **22**).

The algorithm is known to efficiently solve large instances of task mapping problems [85]. However, we further improved it to devise efficient schedules exploiting knowledge about the problem and the reconfigurable architecture onto which tasks are mapped.

2.4.2 Reconfiguration-aware Heuristics

As described in 2.4.1, a 2-step decision process is adopted to reduce the complexity of the exploration, as in [85]. The first one is the *task* local heuristic, which selects, at each decision point, what is the current "best" task to schedule among the set of ready tasks. The second one is the *mapping* local heuristic, which selects, at each decision point, given a task, what is the current "best" choice of processing element and implementation to execute the chosen task with. To balance the relative weight of all the choices, all heuristic values (local and global) are scaled in the $(0, 1)$ interval, where higher means better choices.

The *task* local heuristic yields high values for tasks characterized by low mobility values and relatively fast execution times (averaged over all the task's possible implementations). This rule particularly favors an early execution of ready tasks on the critical path that might otherwise unnecessarily increase the overall execution time. It also slightly increases the heuristic value for tasks that executes relatively faster than the others.

The *mapping* local heuristic, instead, is computed in differing ways depending on the implementation. Moreover, since the architec-

ture might feature static, reconfigurable and software processors, we have to rank the choices featuring any of them in a way that they can be fairly compared to each other. The first step, then, is to understand whether the implementation is hardware or software.

If software, we compute the heuristic as the product of two terms in the $(0, 1)$ interval: H^1_{SW} and H^2_{SW}. H^1_{SW} is the likelihood of a software implementation of the task with respect to *any* hardware one (if available), to take into account how a software implementation might be better or worse than a hardware one. H^2_{SW} computes the *average mobility* of all software processors, as the sum of the mobility values of all the tasks previously assigned to each software processor, divided by the number of those tasks. Software processors that haven't been assigned tasks yet are automatically ranked better than any other software processor choices. This choice privileges software processors that have never been assigned tasks before and that have been previously mapped with tasks with relatively lower average mobility.

If hardware, instead, we consider the set of static IP processors and reconfigurable regions available in the system. For the computation of the heuristic of a reconfigurable region, we compute the product of three terms in the $(0, 1)$ interval: H^1_{HW}, H^2_{HW} and H^3_{HW}. H^1_{HW} is a general likelihood factor for a hardware implementation with respect to *any* software one (if available). H^2_{HW} is a factor that relates to area usage for instantiating the implementation on this reconfigurable region. This term takes into account the number of tasks that are yet to be scheduled in order to assign each task a (potentially) fair share of the resources of the FPGA. We penalize the heuristic values of those choices which would consume more resources (in percent terms) than the percent advancement of the mapping algorithm. For example, if the ant is mapping the 5th task out of 10 available tasks, a good choice must not consume more than 50% of the available FPGA area, after mapping this task to the choice's reconfigurable region. If this limit is violated, but the overall constraint on total FPGA area is not violated, the choice is viable but is consuming more area than it should and so its heuristic value is halved. If instead the violation leads to a global area violation, the solution is discarded (i.e., heuristic value = 0, preventing to take this decision). This allow for an early discovery of unfeasible solutions for faster convergence. H^3_{HW} term is similar to H^2_{SW}: the lower the average mobility of the tasks already assigned to a reconfigurable

region, the better this term. At each iteration, it is required to evaluate the current solution and provide feedback to the exploration heuristic. In this work, we only considered the overall execution time of the application as the metric to be optimized. However, the framework has been designed to accommodate the usage of different metrics (e.g., execution time, power consumption and area occupation) or any combination of them. Furthermore, it is possible to integrate different methods to compute the metrics, ranging from mathematical models to actual simulations, to trade-off elaboration time and accuracy of the evaluation. As an example, the designer can adopt cycle-accurate simulations when analyzing small applications and then simpler heuristics when considering large ones. Also the method adopted for the final validation, i.e., high-level simulation with virtual platforms, can be integrated. In all the cases, the metrics are intended to work on scheduling trace of the DAG produced by the exploration heuristic previously described.

It is worth noting that the scheduling trace only contains the mapping decisions for the tasks on the processing elements; however, given the possibility of exploiting PDR, it is clear that this trace is not enough to determine the application execution time since it does not include the reconfiguration tasks. For this reason, it is then necessary to construct a more detailed representation of the application to be evaluated, starting from this scheduling trace. Note that the transformations that we applied to construct this enhanced representation can be applied to compute any of the metrics mentioned above; however the designer can also apply other transformations to support even more detailed descriptions of the system.

As an example, we consider the trace shown in Table 2.1: it reports an admissible mapping and scheduling for the DAG of Figure 2.1 with respect to the architecture shown in Figure 2.2. It is worth noting that reconfigurations are extracted only when two consecutive tasks are assigned to the same region, but with different implementations. This allows supporting hardware reuse, when multiple tasks are assigned to the same region with the same implementation; in fact, in this case, reconfigurations are not required to switch the functionality.

Starting from this trace, we introduce two more entities in the graph to be scheduled which are the *reconfiguration nodes* and the *communication nodes*. The former represents a reconfiguration that has to be performed to change the functionality of a hardware mod-

Table 2.1: *Example of scheduling trace for the TG in Fig. 2.1*

Task Name	Implementation	Processing Element
T0	A	CPU0
T1	A	RR0
T2	B	RR0
T3	A	RR1
T4	A	RR0
T5	C	CPU0

ule and it is mapped on a dedicated component (i.e., the ICAP on Xilinx devices). The latter, instead, is used to represent the data exchanged directly between tasks or through the memories, according to the communication infrastructure. Given this enhanced representation, we adopted a simple yet effective list-based scheduling algorithm [16] to compute the overall execution time (i.e., the make-span) of the application. The execution time of each task is reported into the implementation determined by the mapping, while reconfiguration and communication overheads are computed as described below.

2.4.3 Reconfiguration nodes

The heuristic described in Section 3.4 assumes the possibility that the hardware modules may have multiple tasks assigned to the same hardware module. Adding reconfiguration nodes consists in identifying where the reconfigurations take place and performing an estimate of their execution time. Starting from the information in Table 2.1, it is possible to determine where reconfigurations occur by identifying which hardware modules implement more than one different implementation. In this example, it is possible to identify reconfigurations only for the module *RR0*, since *RR1* is set to execute only *T3*. Once the reconfigurations have been identified, they have to be accordingly introduced in the task graph. In particular, each reconfiguration is inserted between the task implemented on the hardware core and the one that needs to be reconfigured for the execution on the same module. Furthermore since the scheduling trace also defines the priorities between the tasks, the reconfigurations must respect this order and they can be accordingly ordered. Given the trace in Table 2.1, the reconfiguration nodes identified are reported in Table 2.2 and the resulting task graph is reported in Figure 2.4 (left).

Table 2.2: *Reconfiguration nodes identified starting from the trace reported in Table 2.1.*

Rec. Node	Proc. Elem.	Function	Prev. Task	Next Task	Prev. Rec.
REC0	RR0	B	T1	T2	-
REC1	RR0	A	T2	T4	REC0

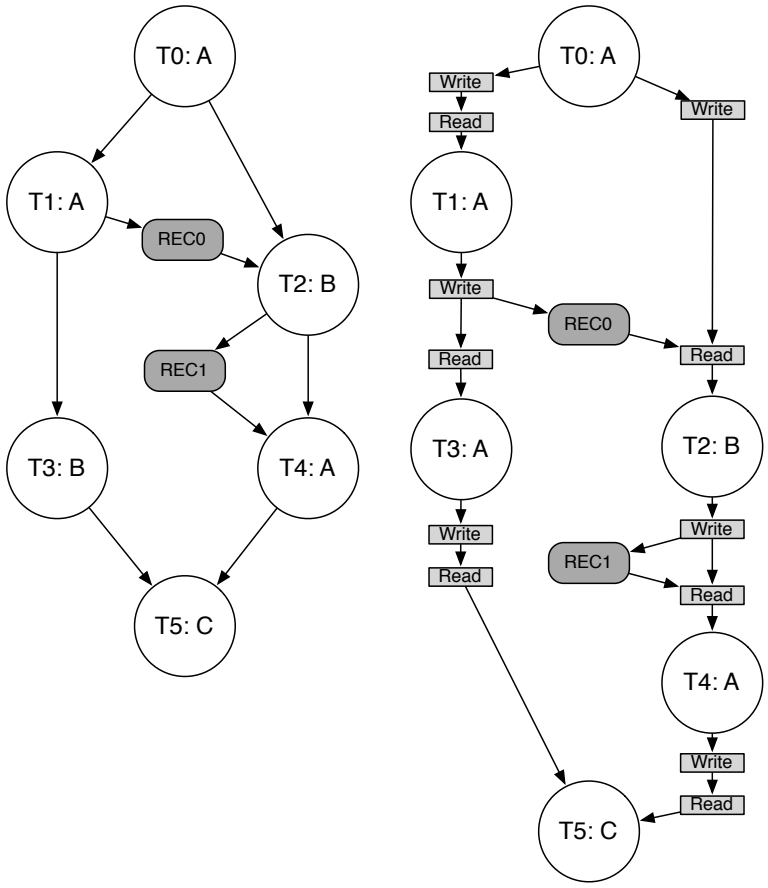


Figure 2.4: *Task Graph of Figure 2.1 extended with reconfiguration nodes (on the left) and with both reconfiguration and communication nodes (on the right).*

Concerning the execution time of a reconfiguration task, we assume that it is proportional to the size of the largest bitstreams of the implementations assigned to the corresponding hardware module. Then, we adopt the same approach proposed in [10] to estimate the actual reconfiguration time required for each region. In our algorithm all the reconfiguration nodes are scheduled to be executed in sequence by a dedicated processor.

2.4.4 Communication nodes

After the reconfiguration, also the communication nodes can be introduced, based on the topology of the interconnection between the modules where the tasks have been assigned. Representing communications as explicit nodes and keeping them separated from the actual computation of the tasks allows to provide, if needed, a more accurate simulation of the overall system behavior. For example, it is possible to include bus congestion metrics during the simulations and envision exploration phases that aim at customizing also the communication architecture for the given application. Considering a bus-based architecture with shared memory, before executing a task, it is necessary to read data from the memory and, after its termination, to send the results to the memory to be used by subsequent tasks. Thus, communication nodes are added before (*read*) and after (*write*) each task and their execution time is proportional to the amount of data to be transferred. The result of integrating communication nodes to the task graph of Figure 2.1 is reported in Figure 2.4 (right).

Regarding the communication time, this is estimated based on the given amount of data to be transferred, as specified in the DAG representation. Note that exploration of data transfers can be also integrated, as in [85], but this is out of the scope of this work.

2.5 Experimental Results

We implemented SMASH in C++ and we then applied the resulting framework to a set of benchmarks generated with TGFF, as in [85]. We then generated the Virtual Platforms (VPs) corresponding to the resulting solutions and we simulated them with Synopsys Platform Architect [192] for rapid prototyping of system-level integration. This allows us to validate the proposed approach in a wide

range of case studies and also to analyze the scalability of the approach. In particular, the generated task graphs are converted to VP models by using the *Generic Task Library*, where their *processing time* is based on the implementation details which the corresponding tasks have been mapped onto. Then, each processing element model is typically available in the Platform Architect library as a Virtual Processing Unit (VPU). VPUs can thus represent all programmable, configurable, or fixed logic processing elements, based on the specified VPU configurations. Also the ICAP controller of the reconfiguration process is modeled as a VPU to execute reconfiguration tasks. Then, modules of other regular IP blocks (e.g., interconnect, memories and DMAs) are instantiated to allow a complete simulation.

The generated task graphs ranges from 10 to 100 nodes, where each task has at least one SW implementation and multiple HW ones, representing realistic trade-offs between execution time and requirement of resources. Large task graphs can also represent multiple applications to be simultaneously executed onto the target platform. The adopted architectural templates (similar to the ones in Figure 2.2) can easily represent embedded systems featuring either soft or hard processors (e.g., Xilinx XUPV5 with Microblazes or the Zedboard's ARM Dual Core Cortex-A9 [25]), augmented with a set of hardware modules. We adopted three architectural templates as starting point for our experiments: *static*, *mixed* and *reconfigurable*. *Static* identifies architectures where the FPGA area is divided into a set of up to k_S static IP cores. *Mixed* identifies architectures where both IP cores and reconfigurable regions are employed to devise a solution, but no more than k_M^{IP} IPs and k_M^R reconfigurable regions may be used at once. *Reconfigurable* represents an architecture with no more than k_R regions. Note that in the second and the last cases, the reconfigurable regions can be also deployed as static cores in the final architecture, in case they are assigned with only one task. We generated two sets of architectures from these templates, based on commercially available Xilinx Zynq-7000 FPGA devices: an Artix-7 and a Kintex-7, with $28,000$ and $125,000$ logic cells, respectively. For the lack of space, we report only the results related to Kintex-7, given that the other ones show a similar behavior.

Figure 2.5 reports the results obtained when executing SMASH for 75 generations with 10 ants for each of them. Indeed, this combination of parameters is an empiric good compromise between greedy search and evolution towards a global optimum based on the pheromone

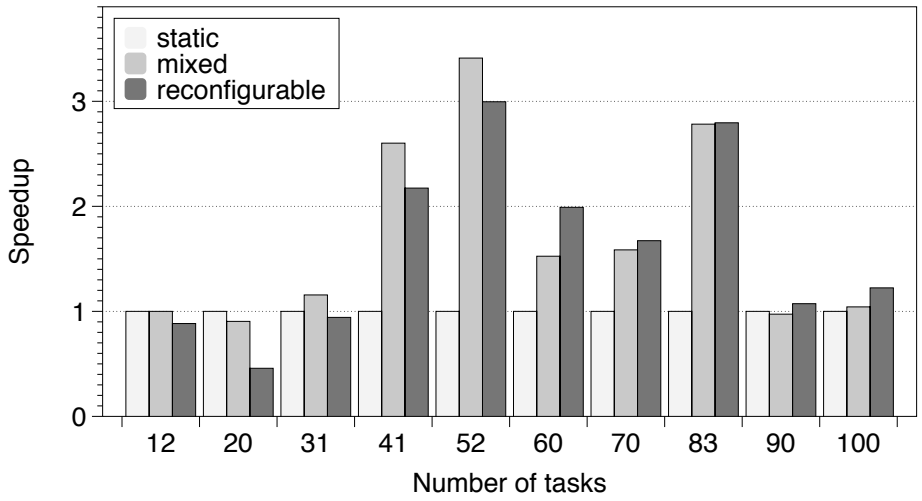


Figure 2.5: *Speedups of mixed and reconfigurable architectures with respect to the static one.*

matrices. Note that, in the graph, we reported the execution times normalized with respect to the fully static execution. Table 2.3 reports instead information about the resulting architectures, along with information about hardware accelerated tasks and reconfigurations.

Results show that SMASH is always able to map a large number of tasks in hardware, generally obtaining relevant speedups when starting from mixed and fully reconfigurable templates. In particular, between 40 to 80 nodes, SMASH is able to obtain speedups up to 3x with respect to static architectures, which limits the number of tasks that can be ported in hardware without exploiting PDR. It is worth noting that, for small instances, reconfiguration can also introduce a slow-down in the execution time since, in this case, its overhead has a proportionally larger impact. On the other hand, larger instances show limited speedups because less tasks can fit in the available area and both software tasks and data transfers can affect the execution time.

The results in Table 2.3 also show that SMASH is effectively able to identify the best combination of static IP cores and reconfigurable regions based on the problem structure. Indeed, when starting from both mixed and fully reconfigurable architectures, it is able to devise which hardware modules require to be reconfigured and, at the same time, determine how to assign and schedule the tasks to mask

Table 2.3: *Results in terms of architectures (number of static IPs and reconfigurable regions - IPs and RRs), along with number of hardware tasks (HW tasks) and required reconfigurations (#Reconf).*

	Static				*Mixed*				*Reconfigurable*			
#Tasks	*IPs*	*RRs*	*HW tasks*	*#Reconf*	*IPs*	*RRs*	*HW tasks*	*#Reconf*	*IPs*	*RRs*	*HW tasks*	*#Reconf*
12	7	0	7	0	7	0	7	0	6	0	6	0
20	20	0	20	0	18	1	20	1	17	1	19	1
31	30	0	30	0	20	4	31	7	16	7	30	7
41	30	0	30	0	18	8	40	14	12	12	40	16
52	30	0	30	0	17	9	51	25	8	17	51	26
60	30	0	30	0	15	10	53	28	10	14	51	27
70	30	0	30	0	17	9	55	28	9	16	58	33
83	30	0	30	0	15	11	80	54	6	19	81	56
90	30	0	30	0	23	3	31	5	9	12	39	18
100	30	0	30	0	16	7	46	23	3	17	53	33

the reconfiguration overhead.

It is also worth noting that SMASH is able to efficiently exploit available hardware resources, occupying always more than 90% of them and never violating the total area constraint.

In conclusion, the results show that the proposed methodology is effectively able to support the designer in the development of reconfigurable architectures, limiting the impact of the decisions performed by the designer about the configuration of the initial architectural template in terms of hardware modules.

2.6 Conclusions and Future Works

In this chaper I presented SMASH, a heuristic and iterative methodology for supporting the design of reconfigurable embedded systems. The methodology have been tested using a set of synthetic task graphs of different size with respect to different architectural templates. It proved to be effective, overcoming the classical limitation in the design of such systems, where the designer is faced with the problem of manually deciding the structure of the architecture.

Future works will focus on two aspects: the first one is the inte-

gration of multiple metrics to optimize multiple objectives as for instance performance and power consumption; while the second consists in the integration of a floorplanning phase to identify also the physical constraints of the resulting modules.

CHAPTER 3

On the Partitioning a Graph of Accelerators

In this Chapter, we further elaborate on the work presented so far hypothesizing that a more general structure of our Streaming Stencil Time-steps (SSTs) could be that of a *graph*. Where it be the case, we would need a mechanism to partition such network onto multiple, interconnected computing elements, i.e. Field Programmable Gate Arrays (FPGAs).

Being similar in nature, we assume that our graph of accelerators can be ideally represented as the result of the manipulation of the code mediated by a process network generator – specifically, a polyhedral process network generator. For this, we assume that our code can be analyzed by PNGen [139, 143–145].

However extensively explored the problem is in the state of art, we could not find a single work that could partition a network using an efficient and effective heuristic partitioning scheme, guaranteeing a fast execution time and the feasibility of the overall partitions. We solve the issue by delineating one such scheme, which is exhaustively reported in this chapter.

3.1 Introduction

Process Networks(PNs)-based models of computation have proven as a successful framework for describing multiple kinds of applications in the Reconfigurable Hardware (RH) domain. Due to their intrinsically parallel and reactive behavior, and well-known techniques to automatically manipulate some of their instances, they are very amenable to Field Programmable Gate Arrays (FPGAs). One problem associated with PNs is that the number of nodes is usually proportional with the parallel portions of computation, and a tool to automatically map tasks to FPGAs is required when multiple FPGAs are employed to improve performance (via increased parallelism). While it is possible to solve this problem in an exact manner via dynamic programming approaches, this is not the case when practical graphs are under examination, i.e. graphs with potentially thousands nodes. In this work we extend a well-known graph partitioning technique, namely Multi-Level K-ways partitioning algorithm, in order to cope with such scenario.

General Partitioning Problem (GPP) plays a major role in data analysis, machine learning, computer science, engineering, and related fields. Most graph partitioning algorithms optimize a ratio between the cut and the size of the partitions, leading to an NP-Complete problem [135]. However, this makes it impractical to partition large networks, which is the reason why an entire field arose to cope with this problem, namely Approximated Graph Partitioning.

Given an un-weighted graph G with V nodes and E edges and given a number K, the Graph Partitioning Problem is to divide the V nodes into K parts such that the number of edges connecting nodes in different parts is minimized given the condition that each part contains roughly the same number of nodes. If the graph is weighted, i.e. the nodes and edges have weights associated with them, the problem considers the sum of the weights of the edges connecting nodes in different parts, while roughly keeping the weights in each partition the same. The problem can be reduced into one where the graph is split into N parts and then merging these nodes to build a smaller graph with fewer nodes, intrinsically easier to partition in the so called initial partitioning phase. In the approximated version of this problem, adequate (possibly random) heuristics are employed to do so [113, 148].

Among the many successful heuristics for partitioning large, highly

interconnected graphs, the Multi-Level Graph Partitioning approach stands apart for both the average quality of the result (i.e.: difference in the resulting final partitions and cut sizes and those generated via the solution of an equivalent optimal problem) and the execution time, usually confined to few minutes on large instances (millions of nodes and arcs) on commodity-level machines. In this approach the graph is recursively contracted to create smaller and smaller graphs which should reflect the same basic structure as the input graph [173]. After that, an initial partitioning algorithm is applied to the smallest graph, in order to obtain a seeding partitioning. Then, each partition of this initial partitioning is further decontracted (un-coarsening) and, at each level, a local search method is used to improve the subsequent partitioning (decontraction/uncoarsening step) induced by the coarser level. The Fiduccia-Mattheyses heuristic for refining the partition after initial partitioning step is employed in this (and other) work to improve the edge cut [113].

Although several successful Multi-Level partitioners have been developed in the last two decades, to the best of our knowledge, none cope with a specific scenarios. Suppose to have a graph (G,V) representing an application. Each node (which we will call *process*) represents a potentially recurrent, potentially periodic task, while edges (which we will call *channels*) represent FIFOs between processes. In this scenario, each process is further characterized by an amount of resources required in order to implement such process p on an FPGA (R_p), and channels are characterized by an amount of sustained data transferred. Additionally, we want to fully exploit this model to compute (i.e. execute processes and data transfers) in parallel, on a multi-FPGA system. In this case, between each FPGA involved in the system, only B_{max} data can be transferred each unit of time, and each FPGA has an amount of resource R_{max}. This is a basic yet accurate representation of the common scenario where a multi-FPGA is designed. In this case, partitioning of the network (for mapping purposes) must take into account how many processes can run onto a single FPGA, and which nodes to map onto which FPGA, in order to cope with given resource constraints. First constraint is related to cut size between each pair of final partitions. In order to meet this constraint we must consider the cut size not only in the original graph but also between each final partition, so that the cut size between each pair of partitions is less or equal to B_{max}. The second constraint is related to resources consumed by each node

(and eventually, by each partition). These two constraints, along with the problem formulation, makes up for the novel contribution of this work.

In this work we present an algorithm that seeks and finds the solution of the Approximated Graph Partitioning Problem in order to satisfy two major constraints that arise when mapping process networks (like Polyhedral Process Networks or Khan Process Networks, to name just a few) onto FPGAs. We use a classical approach to ratio problems where we repeatedly ask whether the solution is greater than or less than some constant which refers to our constraints, based on the Multi-Level Approach.

The rest of the chapter proceeds as follows. As the state of art of GPP is vast, Section 6.2 presents a thorough review of it in order to understand where this work fits and what problem we addressed. Section 3.3 reviews, in particular, the basics of Multi-Level, K-ways partitioning. Section 3.4 describes how we extended previous work in order to cope with the mapping problem at hand. Section 7.5 presents experimental results, and the chapter is ended by Section 3.6 with comments and future work.

3.2 Related Work

There has been a large amount of research on GPP so that we refer the readers to [42, 91, 181] for most of the material. Since finding an optimal partitioning is NP-Complete (and is a well-known, solved problem [97]), one is forced to set up for approximation algorithms in order to find a solution (even though non optimal, in the general case) in a practical amount of time. The part of the investigation in this area concentrates on approaches to solving the Two-Ways Partitioning Problem (TWPP) for bi-sectioning the graph (partitioning the graph into exactly two parts), which is also a NP-Complete Problem. One of the primary attempts and maybe the most well-known heuristic algorithm for partitioning graphs was described in [112], which takes two separate sets as an initial solution of the problem, and trades pairs of nodes between them in order to obtain a candidate solution.

Branch and Bound(B & B) strategy solves the partitioning in the case of $K = 2$, for general weighted graphs have also been presented in [171]. Yan and Hsiao have presented a fuzzy clustering algorithm to solve the GBP and apply it to Circuit Partitioning [215]. Other au-

thors have been presented methods based on *Genetic Algorithms* [48], *Divide & Conquer* approximation algorithms [95] and even *Ant Colony* optimization [49]. *Linear programming*(LP) methods became more popular after being shown that they were able to find better cuts over KL.

Spectral methods additionally got vastly used, since they were faster and produced great results. These are focused on the computation of eigenvectors of the adjacency matrix. Several works have used such techniques like [67, 103, 166]. As an alternative, Multi-Level algorithms for partitioning graphs were initially presented by [31] and [104]. Regularly such Multi-Leveling systems match pairs of adjacent nodes to define new merged graphs and recursively iterate this procedure in order to make a graph with arbitrary nodes. The coarsest graph is then partitioned and the partitions is refined on all the graphs back to the original graph.

Besides to heuristics and approximate algorithms for solving the GPP, many authors have analyzed the lower bounds of known algorithms and in special case of graphs(e.g. [198] and [75]).

Since GP is a hard problem, practical solutions are focused on heuristics. There are two broad categories of methods, Local and Global which we consider here in greater detail.

3.2.1 Local Search Methods

The partitioning can be described as breaking a graph into subgraphs and recursively do it in this way under some constraints in order to make a graph with arbitrary nodes or less than a specific marginal number. Here we will describe this problem by a method known as iterative improvement.

The idea behind iterative improvement is to begin with an initial solution, and make a new solution iteratively until we have a solution that is "good enough". Optimality is measured with respect to a given goodness criteria.

Most iterative improvement techniques are greedy. In a greedy algorithm, the new solution is accepted only if it is better than the old one. Non-Greedy methods (like: hill-climbing algorithms) will sometimes accept a solution that is worse than the existing solution, the reason being that hill-climbing algorithms are used is to avoid getting trapped in local minima. A hill-climbing algorithm can sometimes climb out of a local minimum and find a better solu-

tion by temporarily accepting a solution that is worse than the existing solution.

Two well known local methods in the context of iterative algorithms for GPP are are *Kernighan-Lin* and *Fiduccia-Mattheyses* algorithms, which were the first two-way cuts heuristics adopted by local search strategies. Their significant disadvantage is the arbitrary initial partitioning of the node set, which might have a negative affect on final solution quality. Broadly speaking, given a partition of a graph, a local search algorithm tends to enhance an objective function by moving nodes between partitions. These algorithms let a node move at most once during one iteration of the algorithm. More costly local search algorithms such as Tabu Search eliminate this restriction as far as possible, i.e. a node can be moved different times during one iteration. However, today majority of the methods for enhancing a given partition are variations of the FM algorithm.

Kernighan-Lin Algorithm

The $Kernighan - Lin(KL)$ Algorithm is one the most popular algorithm for the TWPP. *KL* algorithm works as follows:

1. The initial partition is generated Randomly. Create two sub-graphs G_1, and G_2. If the graph has N nodes, the first $\frac{n}{2}$ are assigned to G_1, and the rest are assigned to G_2.

2. A solution is acceptable only if both sub-graphs contain more or less the same number of nodes.

3. The goodness of a solution is equal to the number of graph edges that are cut between partitions.

4. The technique for generating new solutions from old solutions is to select a subset of nodes from G_1, and a subset of nodes from G_2 and swap them. To maintain acceptability, we always select two subsets of the same size.

KL drawbacks are:

1. handling of unit node weights only,

2. handling of exact bi-sections only,

3. time complexity of a pass is high, $O(n^3)$.

Fiduccia-Mattheyses Algorithm

There have been great improvements made to the KL algorithm. The most imperative change is a slight adjustment of the algorithm and the decrease in running time that was provided by $Fiduccia-Mattheyses(FM)$ [87]. Fiduccia and Mattheyses suggested following modifications:

1. Only one node is moved at a time,
2. The consecutive moves are made in the opposite directions,
3. The algorithm maintains a sorted list of candidate interior nodes for moving to the other sub-graph, and updates it after each move.

They succeed to decrease the complexity for a single pass to $O(n)$ by using modern data structures. Like the KL strategy, the FM strategy performs passes where each node is moved at most once, and the best bi-section observed during an iteration is used as input for the next iteration. In any case, instead of selecting pairs of nodes, the FM method chooses just single nodes for moving. *Fiduccia-Mattheyses* balanced the algorithm and adopted adequate date structures such that the asymptotic running time of their local search algorithm is reduced to linear time $O(n)$.

3.2.2 Global Search Methods

Global search relies on the properties of the entire graph and do not rely on an arbitrary initial partition.

One such technique (specifically aimed at solving the TWPP) is to formulate it as a quadratic optimization problem. However, due to the nature of the optimization problem, realistic graphs still result unmanageable. For this reason, a class of graph partitioning methods, called Spectral Methods – the most common example of which is Spectral Partitioning, where a partition is derived from the analysis of the spectrum of the adjacency matrix – relax this optimization. Spectral techniques have been enhanced by several works like [47, 151]. Unfortunately, to the best of our knowledge, none of the previous methods contemplate the partitioning of applications in the presence of simultaneous resource and bandwidth problem constraints (or the equivalente in the respective formulations).

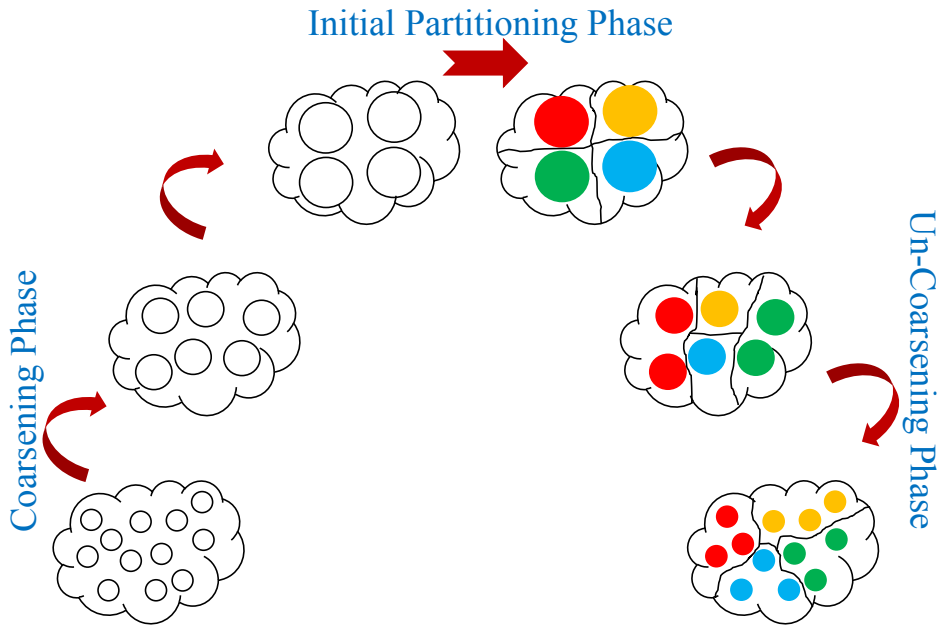


Figure 3.1: *Multi-Level Scheme [24]*

Other methods contemplate Multi-Way Spectral Bisection Algorithm and Parallel Graph Partitioning [30, 98] and Multi Level, K-Ways Partitioning. As this last technique is the basis for this work, we detail the inner workings in the following Section.

Previous work – as presented in this brief recall of the state of art – focuses on heuristically minimizing the cut size associated to the partitionings found. *However, as the techniques focus on such minimization, to the best of our knowledge, none address the problem that we approach in this work: a cut size minimization algorithm with novel constraints tightly related to the reconfigurable hardware domain.*

3.3 Multi-Level, K-ways Partitioning

[104] formulated this strategy as it is known now. The Multi-Level approach to GP comprises of three main phases, which are reported in Figure 3.1.

In the **contraction (coarsening)** phase, a hierarchy of graphs is created. The most common way to build this hierarchy is to itera-

tively identify matching $M \subseteq E$ and contract the edges in M. Contraction should rapidly reduce the size of the input and each computed level should reflect the global structure of the input network.

Contraction is halted when the graph is small enough to be directly **partitioned** using some costly other algorithm like the ones described in the previous Section (such as KL, FM algorithms and spectral partitioning).

In the **un-coarsening** phase, matching nodes and arcs – which had been previously been merged together in the coarsening phase – are iteratively un-coarsened.

During un-contraction of matching graphs, a local improvement algorithm moves nodes between partitions to enhance the cut-size or balancing constraint. Generally variants of the *FM* algorithm are used. The vision behind this technique is that a good partition at one level will also be a good partition on the next finer level, so that local search will rapidly find a good solution. Moving a node on a coarse level hierarchy typically corresponds to the movement of a whole set of node movements of the finest level of the hierarchy. Intuitively, the Multi-Level scheme has a global view on the optimization problem on the coarse levels of the hierarchy and a very local view on the finest levels with respect to the primary one.

[103] is the first work to report a linear time $O(n)$ implementation of this scheme to obtain $K - Partitions$ (using Recursive Multi-Level Bi-section only on the coarsest level and a direct $K - Way$ local search algorithm). A variant of the Multi-Level algorithm has been proposed in [148]. Their $n - level$ approach is based on the extreme idea of contracting only one single edge between two consecutive levels of the Multi-Level hierarchy. During un-coarsening, local search is done highly localized around the un-constructed edge. Using complicated data structures their algorithm requires sub-linear time on real graphs.

Compared with Multi-Level Spectral Bisection, Multi-Level K-Way partitioning is usually two orders of magnitude faster, and produces partitioning with generally smaller edge-cuts. This is why we employed this basic scheme for the implementation of our partitioning algorithm, which is described in Section 3.4.

3.4 Algorithm's Internals

The proposed method is based on a variant of the aforementioned Multi-Level, K-Ways Partitioning (MLKWP) scheme.

In the proposed algorithm, the input graph is coarsened to a parametrized size (default is 100). However, it is not un-coarsened and refined back to the original graph in just one step. Rather, it is un-coarsened up to a certain intermediate level and then coarsened back to the lowest level if needed. This process of un-coarsening and refining up to an intermediate level and coarsening again to the lowest level is repeated a number of parametrized times, depending on whether we are already meeting the constraints or not.

At each iteration, we generate different intermediate clusterings, that are compare *a posteriori* using a goodness function; the best (i.e. the one that is nearest to meeting the constrains) is chosen as the "correct" intermediate un-coarsening candidate. This step incentives rapid convergence while accounting for broad exploration of different clusterings.

After the coarsening phase, we try to meet the K different partitions with the help of initial partitioning phase.

3.4.1 Coarsening Phase

In the coarsening phase we use three type of different matchings in order to better explore different results given multiple search strategies:

- *Random Maximal Matching*,
- *Heavy Edge Matching*,
- *K-Means Matching*.

Random Maximal Matching Nodes of a graph are randomly visited. If there is a node u which is not matched, then one of its unmatched neighboring nodes is randomly selected. Two nodes are said to be adjacent if there exists an edge that is incident to those two nodes. If there exists such a node v, the edge (u, v) is included in the matching and the nodes u and v are marked as matched. Node u remains un-matched in the random matching if there is no unmatched adjacent node v. The goal in the GP is to minimize the sum of the weights of the edges between the nodes on the boundary of

the parts of the graph. Using a randomized algorithm, a maximal matching can be found so a randomized matching method may not always produce satisfactory results for every graph. In order to decrease the edge cut value, heavy edge matching [113] can be used.

Heavy Edge Matching As the name suggests, the edges are sorted according to their weights and matching begins by selecting the heaviest edge. All the edges are visited in descending order and edges with un-matched end points are selected. This heuristic is used when the graph size has been reduced substantially so that not much work is done in sorting the edges.

K-Means Matching Clusters are formed on the basis of their weight; a subset of near nodes is chosen accordingly.

The main objective and theme of this method is to divide the graph into smaller partitions and based on the concept that it first divides the problem into multiple sub-partitions by dividing the total number of the nodes by the number of sub problem you want and assign the nodes to the partitions which is near to the specific cluster [114].

We use in this work all three heuristics algorithms (*Random*, *HEM*, *K-Means*) to get the matching. These heuristics are employed at different times, multiple times, in order to find the best matching for the given graph. Each time we compare the results of the three heuristics with each other and choose the best one.

Once we obtain the matching of nodes to coarsened graphs, we create a map from the nodes in the un-coarsend graph to those in the coarsened graph. Then, using the matches and the map, the coarser graph is built, ready for the next iteration of the coarsening step. Thea adjacency matrix of the coarsened graph is adjusted according to the new incidence between coarser nodes in the graph. The edge weights, in particular, are all copied over but when the matched nodes have a common neighbor: in this case weights are merged into one and the new edge has a weight equal to the sum of the weights of the merged edges. Similarly, the new node gets the sum of the weights of the merged nodes. Any duplicate edge resulting from the process is merged together with their weights added.

The coarsening phase of our algorithm continues until few nodes remain (for example 100 nodes – this is a parameter in our implementation). The resultant most coarsened graph is considered an initial partitioning for the initial partitioning phase.

3.4.2 Initial Partitioning Phase

After reducing the original graph into multiple sub-partitions we produce an initial partitioning of it, with a number of partitions much lesser than the required one.

We adopt a greedy approach, as it is a heuristic that usually yields good results. Specifically, we partition the graph in such a way that we have a balanced number of resources in each part (note how balancing resources is not a priority in our case, while meeting the resource and bandwidth constrain is). After that, we check the bandwidth between each pair of partitions and use the FM algorithm to meet the bandwidth constraint.

We start off with the heaviest nodes. After finding the heaviest one, we'll take it in the first partition among K partitions available and add its neighbors (which are connected via edges to this node) as long as the total number of resources assignable to each partition (R_{max}) is not violated. After this we apply the same for the other partitions as far as all nodes assigned to exactly one partition. Since this method is sensitive to the initial node selection, the whole process is repeated with a parametrized number of randomly chosen initial nodes (10 is default). Since the coarsest graph is no more than a few hundred nodes (100 is our default), running this algorithm K times does not add much to the total partitioning cost. The final partitioning that gives the best cut-size is returned.

After this allocation we pay attention to the remaining nodes (if any) which are not assigned to any partitions. First we try to put each remaining node in accordance to its resources to the first partition which has biggest free space for that node and do it for all remaining nodes. If after this step there are still nodes to assign, we assign each node to the partition which has the biggest free space even though this implies violating the R_{max} constraint. After this step we check the Bandwidth between each pair. If it doesn't meet the constraints we use an *FM*-based algorithm to minimize it as far as possible. Partitions will be changed and nodes will move between partitions as far as constraints met.

We then un-coarsen as necessary, as described in the next Section, in order to obtain the right number of partitions, each meeting the constraints.

3.4.3 Un-Coarsening Phase

During the un-coarsening (refining) phase, the initial partition of the coarsest graph is projected onto the lower level, finer graph. This procedure is repeated until a partition is projected onto the top level graph and is refined to obtain the final partition and cut-size and resource allocation for the graph. The mapping vector is used to project the coarse graph partition onto the finer graph. But if we do not met constraints, we go back to coarsening phase and then partitioning phase (randomly), cyclically. If after a predetermined number of iterations a feasible partitioning is still missing, a message will signal that partitioning with these constraints is either impossible or we have to give the tool more time (i.e.: iterations) to compute such solution.

3.5 Experimental Results

We compare *METIS* and *GP* using random generated graphs. We employ particularly small instances in the following part of this Section in order to visualize the different behavior of the two tools when partitioning the given networks. In all cases, these graphs represent Process Networks generated via suitable tools. Each *process* (i.e.: node) is characterized by an amount of resources required to implement such process on an FPGA (only one resource is considered at this time, for example LUTs) and each *channel* (i.e.: edge) is characterized by an amount of bandwidth consumed. Only bandwidth outgoing from and incoming to different partitions consume bandwidth – we assume that there is enough bandwidth on the FPGA to sustain enough computation between nodes belonging to the same partition (i.e.: FPGA).

We synthetically generated few graphs with the following goal in mind: *to demonstrate that GP can always partition the given network while respecting resource and bandwidth constraints (or fail while doing so) while METIS always partitions, regardless of said constraints*. Graphs are represented as incidence matrices, and are given as inputs to MATLAB.

Figure(5.1)~Figure(5.3) compare the results obtained running both *METIS* and *GP*. Various *GP* parameters are used across all experiments. For *METIS*, we used the default parameter values and decode the results in Matlab in order to compare the results with *GP*.

We compare:

1. Local Edge Cut (i.e. bandwidth insisting between each pair of partitions),
2. Maximum Resources Allocation (i.e. the maximum amount of resources consumed by all partitions),
3. Algorithm's Execution Runtime,
4. Global Edge Cut Sum.

The machine we employed is a 2.53 GHz Intel(R) Core(TM) i5-M 460 CPU with 8GB RAM running Ubuntu14.0464bit. The code runs under MATLABR2013a. METIS5.1.0 is used for comparisons. We refer to the Graph Partitioner of this work as *GP*.

Experiment 1 We consider a graph with 12 nodes and 33 edges for the first experiment. Maximum bandwidth constraint is 16 units. Maximum resources constraint is 165 units.

As it is possible to see in experiment 3.1 – red font – *METIS* violates both constraints while *GP* meets both of them. However, the size of the cut is slightly bigger for *GP*, which is a consistent result as *METIS* tries to minimize the overall cut, but generally violating bandwidth and/or resource constraint. *GP* does not violate any partition-to-partition bandwidth constraint, but it fails at *globally* minimizing the edge cut. Actually, it does, under the bandwidth constraint.

Figure 2 reports the unpartitioned graph (radius of nodes proportional to weight), Figure 3 the same graph with weight and edges allocation, Figure 4 partitioning with *GP*, and Figure 5 partitioning with *METIS*.

Experiment 2 In Figure II we consider a graph with 12 nodes and 30 edges for the second experiment. We apply the following constraints: 25 for bandwidth and 130 for resources. *METIS* violates bandwidth while meeting (incidentally) resources, while, again, *GP* meets both of them.

Incidentally, the local refinement strategy employed translates, in this graph, in a better overall global cut, as reported in Figure 3.2.

Figure 6 reports the unpartitioned graph (radius of nodes proportional to weight), Figure 7 the same graph with weight and edges allocation, Figure 8 partitioning with *GP*, and Figure 9 partitioning with *METIS*.

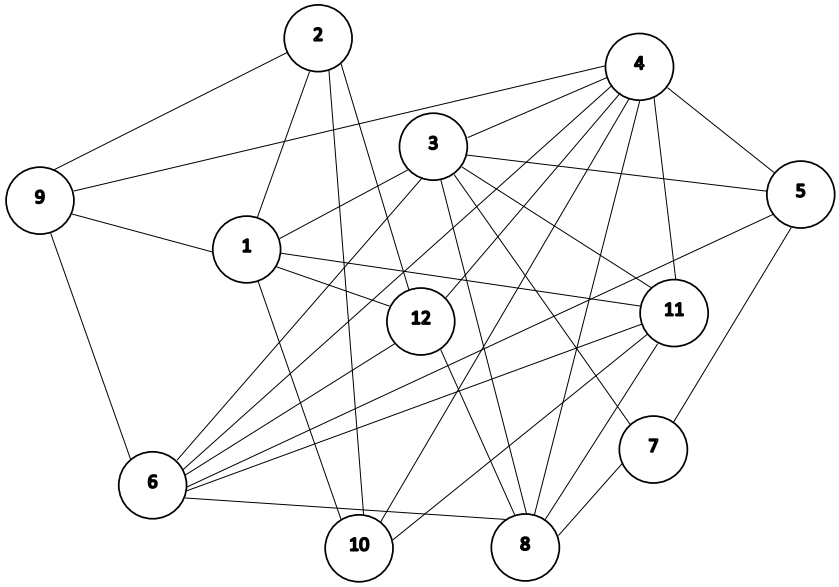


Figure 3.2: *Un-partitioned sample graph 1 before weighting and resource allocation.*

Experiment 3

In the last experiment, whose data are shown in Figure III, we consider a graph with 12 nodes and 32 edges. I apply the following constraints: 20 for bandwidth and 78 for resources. *METIS* violates bandwidth while meeting (incidentally) resources, while, again, *GP* meets both of them.

Figure 10 reports the unpartitioned graph (radius of nodes proportional to weight), Figure 11 the same graph with weight and edges allocation, Figure 12 partitioning with *GP*, and Figure 13 partitioning with *METIS*.

3.5.1 Summary

As it is possible to see from experimental Figures *GP* can always (on the test cases) partition without violating given constraints, which is not guaranteed to be true with *METIS*. Additionally, in our test cases the increase in cut size is near to negligible; however, this might not be the case if we employed stricter constraints.

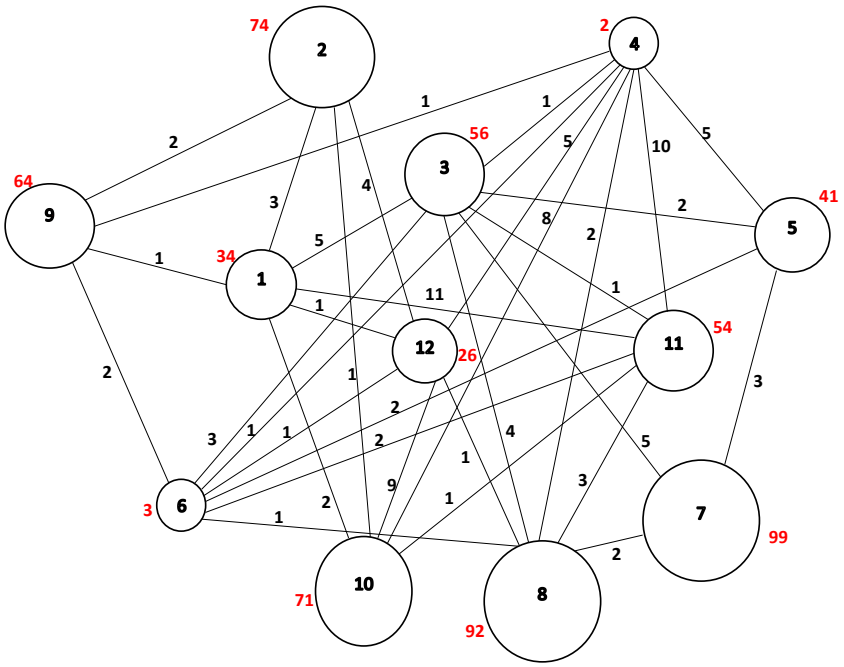


Figure 3.3: *Un-partitioned sample graph 1 after weighting and resource allocation*

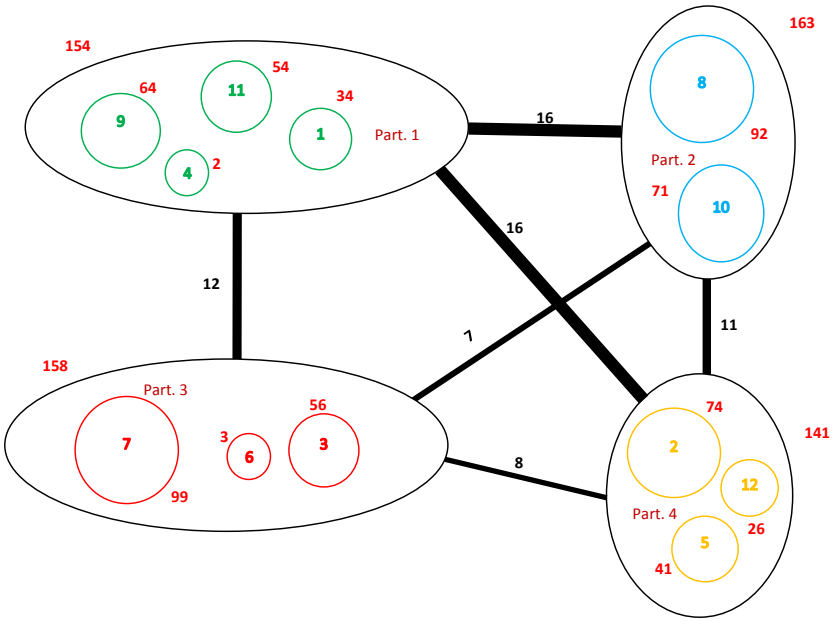


Figure 3.4: *Partitioning of the sample graph 1 with GP algorithm, both constraints are met, constraints are : bandwidth = 16 and resources = 163.*

3.6 Conclusions

We presented a novel approach to partitioning a process network in the presence of simultaneous bandwidth and resource constraints, based on the Multi-Level, K-Ways approach already known in literature. We developed a tool that extends METIS in that it copes with situations where partitioning must happen within precise bandwidth and resource constraints. Future work contemplates the test of this system on actual multi-FPGA based systems where the mapping of potentially large application graphs (process networks) is a difficult task to do by hand.

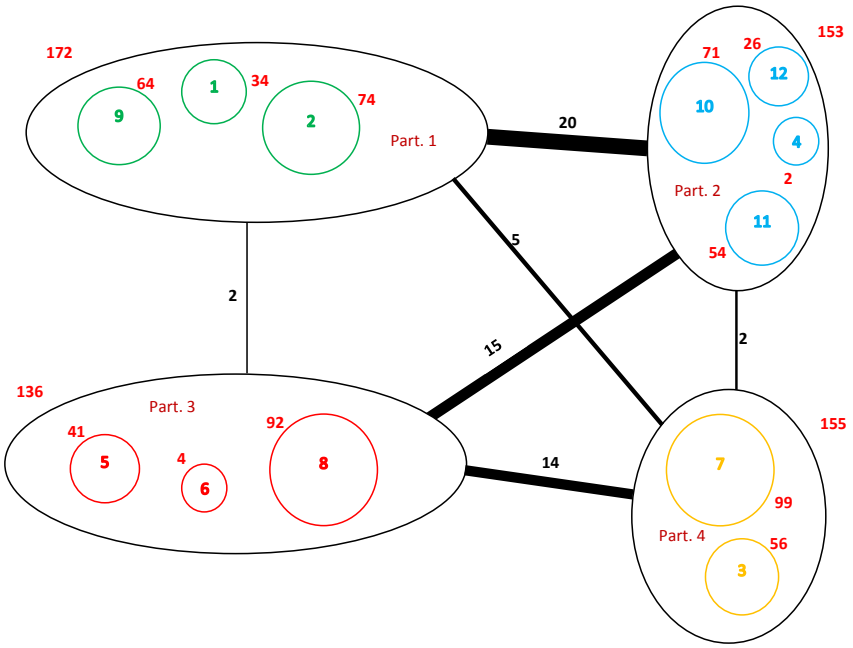


Figure 3.5: *Partitioning of the sample graph 1 with METIS algorithm, both constraints are violated, constraints are : bandwidth = 16 and resources = 163.*

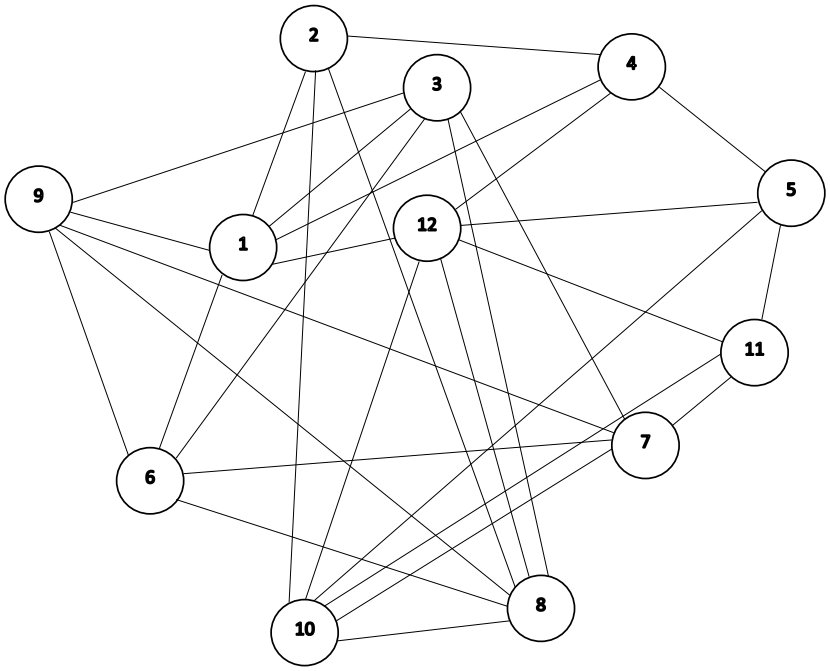


Figure 3.6: *Un-partitioned sample graph 2 before weighting and resource allocation.*

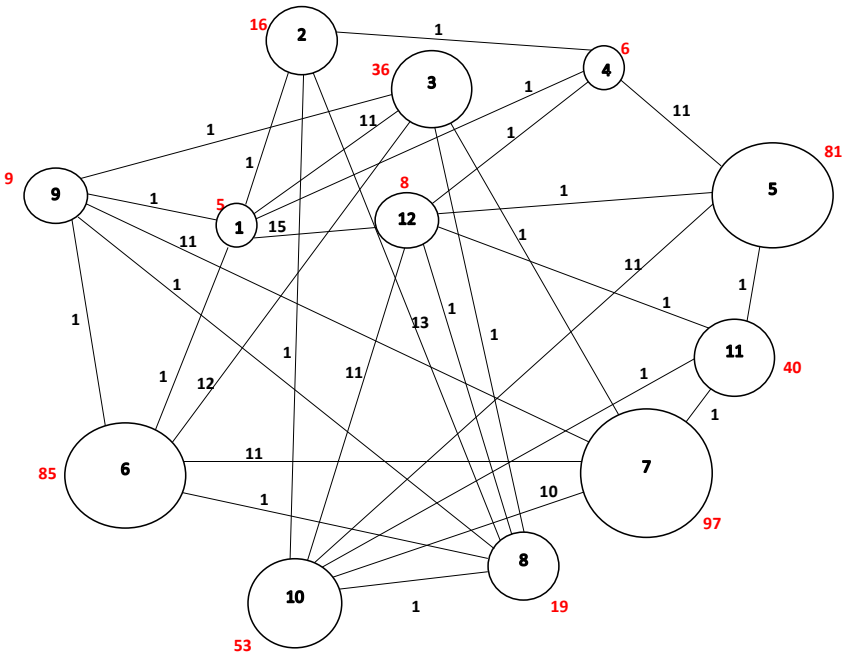


Figure 3.7: *Un-partitioned sample graph 2 after weighting and resource allocation.*

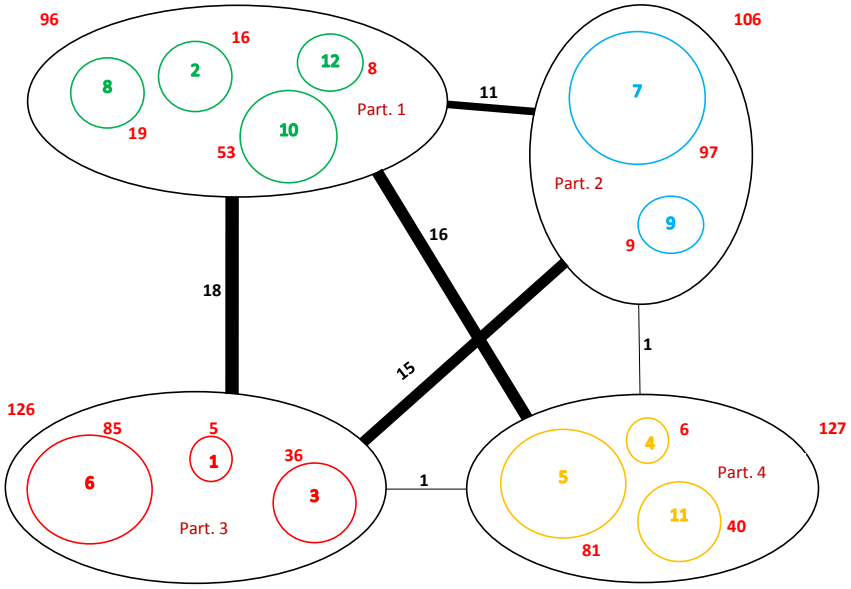


Figure 3.8: *Partitioning of the sample graph 2 with GP algorithm, both constraints are met, constraints are : bandwidth = 25 and resources = 130.*

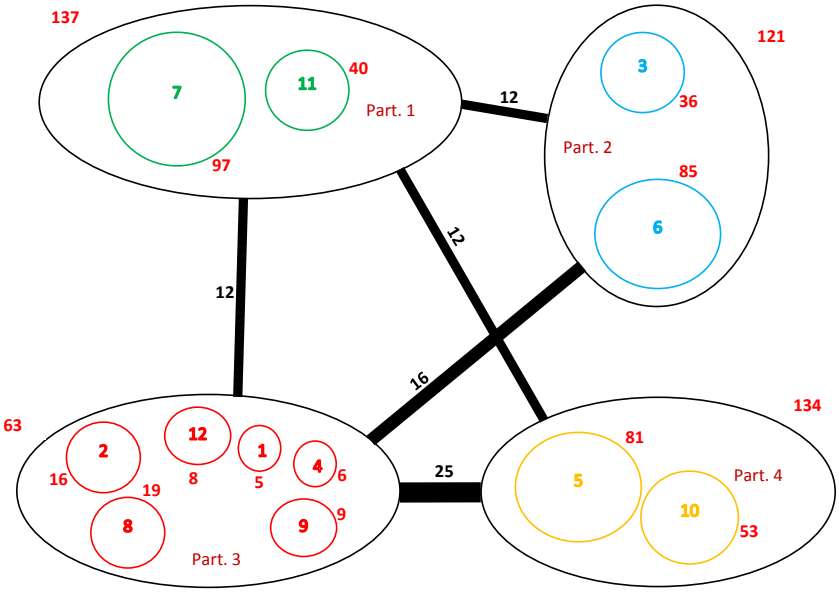


Figure 3.9: *Partitioning of the sample graph 2 with algorithm, resources is violated while bandwidth is met, constraints are : bandwidth = 25 and resources = 130.*

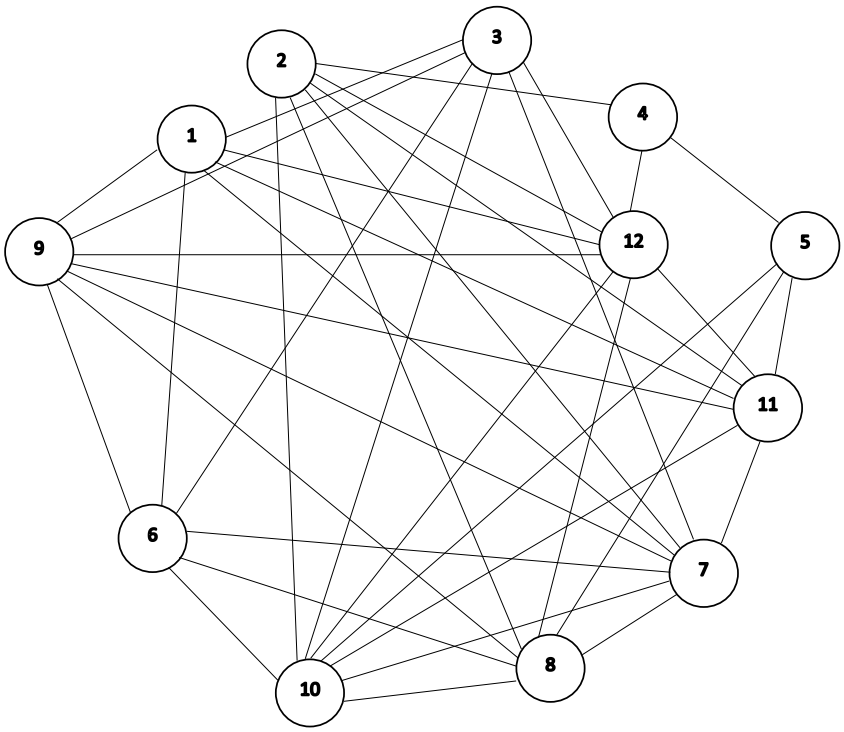


Figure 3.10: *Un-partitioned sample graph 3 before weighting and resource allocation.*

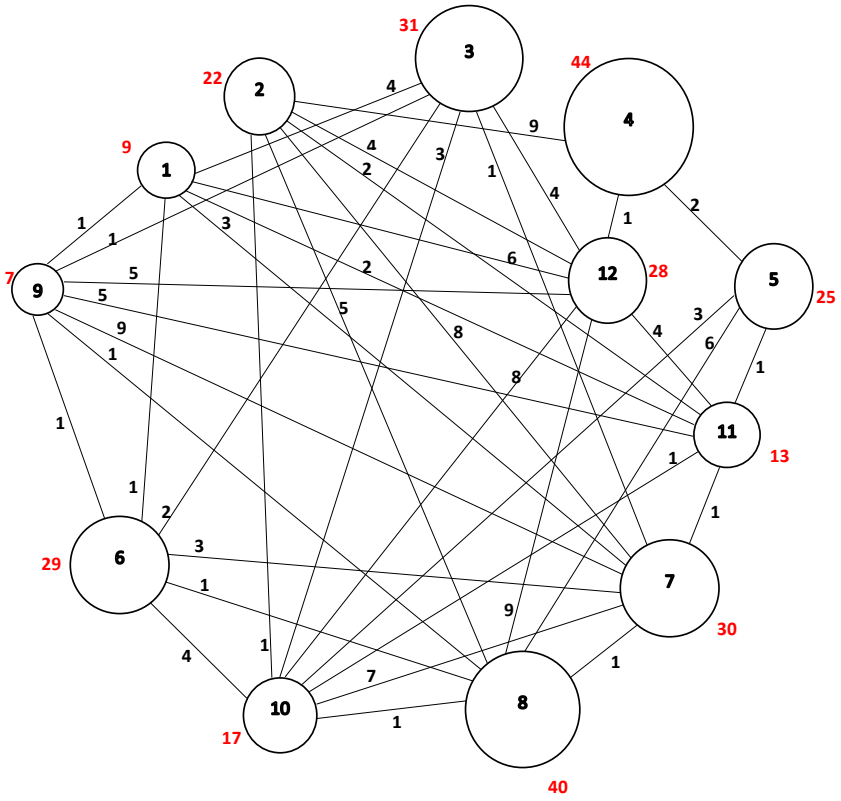


Figure 3.11: *Un-partitioned sample graph 3 after weighting and resource allocation.*

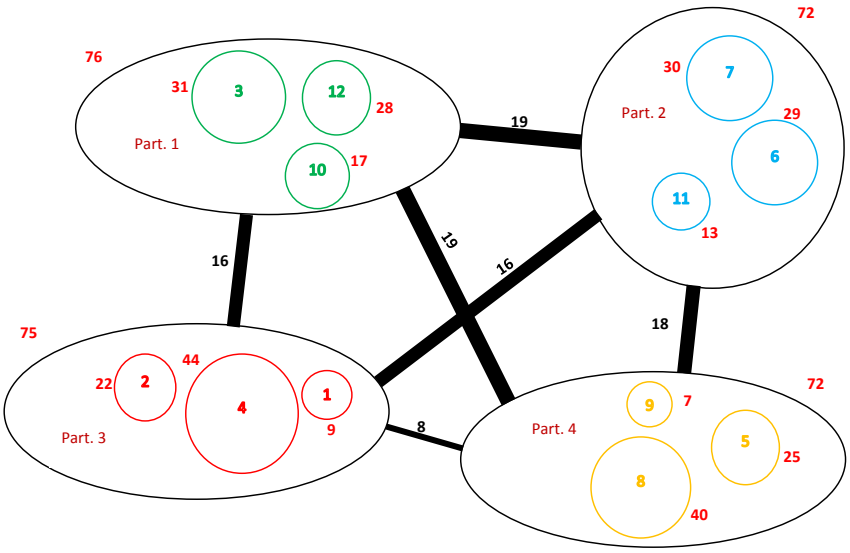


Figure 3.12: *Partitioning of the sample graph 3 with GP algorithm, both constraints are met, constraints are : bandwidth = 25 and resources = 130.*

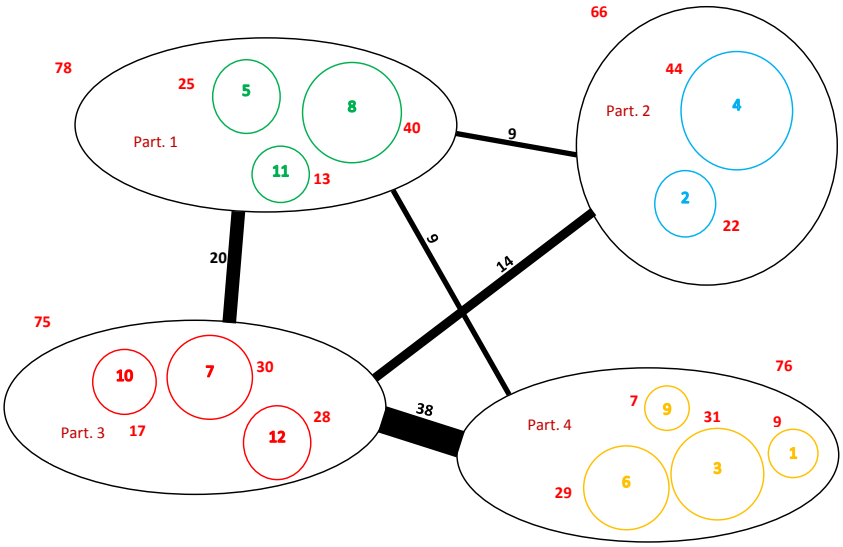


Figure 3.13: *Partitioning of the sample graph 3 with METIS algorithm, resources is violated while bandwidth is met, constraints are : bandwidth = 25 and resources = 130.*

	K = 4			
Algorithms	Total Edge-Cuts	Total Time(S)	Maximum Resource Allocation	Maximum Local bandwidth
METIS	58	0.02	**172**	**20**
GP	70	0.33	163	16

Table 3.1: *Number of Nodes = 12, Number of Edges = 33, both constraints are violated in METIS and in GP both constraints are met.*

	K = 4			
Algorithms	Total Edge-Cuts	Total Time(S)	Maximum Resource Allocation	Maximum Local bandwidth
METIS	77	0.02	**137**	25
GP	62	0.25	127	18

Table 3.2: *Number of Nodes = 12, Number of Edges = 30, resource is violated in METIS but bandwidth is met and in GP both constraints are met.*

	K = 4			
Algorithms	Total Edge-Cuts	Total Time(S)	Maximum Resource Allocation	Maximum Local bandwidth
METIS	90	0.02	78	**38**
GP	96	7.76	76	19

Table 3.3: *Number of Nodes = 12, Number of Edges = 32, bandwidth is violated in METIS but resource is met and in GP both constraints are met.*

CHAPTER 4

A Review of The Polyhderal Analysis Framework

In order to more easily read through the rest of this thesis, a thorough understanding of the background theoretical concepts is required. In this chapter I reconsider the technologies, methodologies and concepts that are relevant to this work and that qualify as state-of-the-art. In Section 4.1 the Polyhedral Model (PM) is presented, along with some use cases like how it is employed to provide automatic information extraction from the input program. In Section 4.4 I show the set of limitations imposed on the code to be analyzed by Polyhedral Analysis (PA), while in Section 4.2 Streaming-based Systems on Field Programmable Gate Arrays (FPGAs) are introduced, since the proposed architecture is streaming-based as well. Section 4.3 is instead dedicated to High Level Synthesis (HLS), because it is the technology that in this work allows to connect the PM with the generation of the architecture, and nonetheless substantially ease the hardware design, also enabling automation. Finally, in Section 4.5 focus is on Iterative Stencil Loops (ISLs), which are

indeed the target of this entire work.

4.1 Introduction to Polyhedral Framework

In scientific and engineering applications, but in general in the great part of computationally intensive programs, most of the execution time is spent in nested loops. This implies that the ability to perform loop nest restructuring towards optimization and parallelization is mandatory, although undoubtedly non-trivial. Standard compilers use in fact Intermediate Representations (IRs) such as *syntax trees*, *call trees*, *control-flow graphs* which are simply not appropriate to perform such a task, as the kind of abstraction of those techniques inevitably hides certain properties and features of programs, making it impossible to perform complex code transformations.

These limitations have created the need to develop techniques specifically aimed at optimizing loop nests, to be used in place of or in combination with standard compilers. A first attempt in this direction has been made in the eighties [162, 214], motivated by the need to map parallel computations onto *systolic arrays* [130]. It was based on the work of Karp et al. [111] that proposed a mathematical model which mapped onto uniform recurrence equations, which also inspired a series of fundamental works from Feautrier [79–83], arrived in the late eighties as well and quickly followed by other works related to the same topic, such as [22, 210, 211]. Those works provided a robust mathematical framework for *regular* imperative programs, and gave the basis to which is now known as the PM (sometimes called *Polytope Model*). The proposed model rapidly evolved and gained importance, as it allowed to map programs onto a mathematical representation, creating a solid link with algebra, as well as Operations Research (OR), thus making possible to extend their applicability also in the field of programs optimization. With the aid of the Polyhedral Model, loop optimization has reached the point in which a finely calibrated transformation can condense in a single step the equivalent of a significant number of textbook loop transformations [19, 99].

In the following section, a detailed overview of the polyhedral framework is then provided, starting from the model, described in section 4.1.2, and explaining what can be accomplished with such a model, which is the topic of section 4.1.3.

4.1.1 Motivating Examples

With the purpose of showing the PM usefulness, let us consider this *matrix multiplication* algorithm as a motivating example:

```
for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
        for (k = 0; k < nk; ++k)
            C[i][j] += alpha * A[i][k] * B[k][j];
```

This loop nest, as simple as it might seem, is indeed not optimized at all. In fact, just by considering a simple memory system, featuring only a single level of cache, it is obvious that the execution suffers a slow down due to cache misses, since every instance of the statement requires in fact two concurrent reads, both from A and B, resulting in no utilization of the cache. However, a simple restructuring completely changes the situation:

```
for (i = 0; i < Ni; i++)
    for (j = 0; j < Nj; j++)
        for (k = 0; k < Nk; ++k)
            C[j][k] += alpha * A[j][i] * B[i][k];
```

Just by changing matrix indices, the execution is now optimal with respect to *data locality*, since now a new cell of A is read only at every new iteration of the outer loop, allowing the maximum data reuse available.

Let us now consider the following code:

```
for (i = 1; i < ni; i++)
    for (j = 0; j < nj; j++)
        A[i][j] = A[i-1][j] + 1;
```

As it is, this loop cannot be parallelized among the outer loop. The reason for this situation is the dependency between the point (i, j) of the loop instance and the previously computed point $(i - 1, j)$, that, due to the way in which the array is traversed, enforce a dependency between instances among the outer loop.

However, there is a simple solution to this problem: by changing the way in which the loop nest is executed, the dependency scheme can be exploited to enable the parallelization of the outer loop.

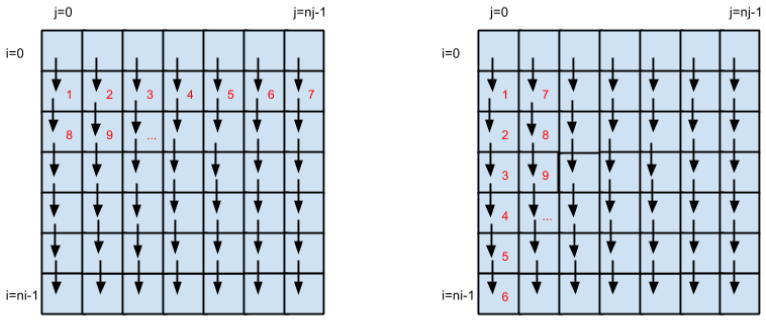


Table 4.1: *On the left, the numbers show the traversing order of the original code. The traversing order enforced after the transformation is instead shown on the right. The array dependencies are represented with an arrow.*

```
for (j = 0; j < nj; j++)
    for (i = 1; i < ni; i++)
        A[i][j] = A[i-1][j] + 1;
```

The loop nest can now be partitioned among different, *parallel*, computing units.

As last example, consider the following loop nest:

```
for (i = 1; i < ni; i++){
    A[i] += A[i-1];
    B[i] += B[i-1];
}
```

The two statements are *completely independent*, hence, they could in principle be computed separately. The code is however written in a way that does not allow such optimization. Splitting the loop in two solves the problem,

```
for (i = 1; i < ni; i++)
    A[i] += A[i-1];

for (i = 1; i < ni; i++)
    B[i] += B[i-1];
```

allowing now to execute the two loops in parallel.

This is indeed exactly the kind of transformations that are enabled by the polyhedral framework.

4.1.2 Polyhedral Model

The PM has been proved to be a powerful tool for automatic optimization and parallelization. In fact, at the price of certain regularity conditions, this model can deliver very high standard in terms of execution time, throughput, number of processors and communication channels, memory requirements, and so on. It is indeed based on an algebraic representation of programs, whose manipulation allows to construct and search for complex sequences of optimizations.

This section precisely describes this model, giving a comprehensive overview of all the building blocks.

Mathematical Background

In order to understand the following concepts, this section provides the key definitions for polyhedral theory, the mathematical background on which the PM rests its foundations [154].

Definition 1. Convex Set. *Given S a subset of $\mathbb{R}^n$. S is convex iff, $\forall \mu, \lambda \in S$ and given $c \in [0,1]$:*

$$(1-c).\mu + c.\lambda \in S$$

A set is convex if for every pair of points within the object, drawing a line segment that joins the pair of points, each point on this segment is also in the set.

Definition 2. Affine Function. *A function $f\colon \mathbb{K}^m \to \mathbb{K}^n$ is affine if there exists a vector $\vec{b} \in \mathbb{K}^n$ and a matrix $A \in \mathbb{K}^{m\times n}$ such that:*

$$\forall \vec{x} \in \mathbb{K}^n,\ f(\vec{x}) = A\vec{x} + \vec{b}$$

Definition 3. Affine Spaces. *A set of vectors is an affine space iff it is closed under affine combinations.*
A line in a vector space of any dimensionality is a one-dimensional affine space.

Definition 4. Affine half-space. *An affine half-space of $\mathbb{K}^m$ (affine constraint) is defined as the set of points:*

$$\{\vec{x} \in \mathbb{K}^m |\ \vec{a}.\vec{x} \leqslant \vec{b}\}$$

Definition 5. Affine hyperplane. *An affine hyperplane is an $m-1$ dimensional affine sub-space of an m dimensional space.*

An hyperplane divides the space into two half-spaces*, the positive and negative half-space. Each half-space can be represented by an affine inequality.*

Definition 6. Polyhedron*. A set* $S \in \mathbb{K}^m$ *is a polyhedron if there exists a system of a finite number of inequalities* $A\vec{x} \leqslant \vec{b}$ *such that:*

$$\mathcal{P} = \{\vec{x} \in \mathbb{K}^m | \, A\vec{x} \leqslant \vec{b}\}$$

Equivalently, it can be defined as the intersection of finitely many half-spaces. Hence the representation as above, where each inequality corresponds to a face of the polyhedron.

Definition 7. Parametric Polyhedron*. Given* $\vec{n}$ *the vector of symbolic parameters,* $\mathcal{P}$ *is a parametric polyhedron if it is defined by:*

$$\mathcal{P} = \{\vec{x} \in \mathbb{K}^m | \, A\vec{x} \leqslant B\vec{n} + \vec{b}\}$$

Definition 8. Polytope*. A polytope is a bounded polyhedron.*

Definition 9. Integer Hull*. The integer hull of a rational polyhedron* $\mathcal{P}$ *is the largest set of integer points such that each of these points is in* $\mathcal{P}$*.*

Definition 10. Lattice*. A subset* L *in* $\mathbb{Q}^n$ *is a lattice if is generated by integral combination of finitely many vectors:* $a_1, a_2, ..., a_n (a_i \in \mathbb{Q}^n)$*.*

$$L{=}L(a_1, ..., a_n) = \{\lambda_1 a_1 + ... + \lambda_n a_n | \lambda_i \in \mathbb{Z}\}$$

If the a_i *vectors have integer coordinates,* L *is an integer lattice.*

Definition 11. $\mathbb{Z}$-polyhedron*. A* $\mathbb{Z}$*-polyhedron is the intersection of a polyhedron and an affine integral full dimensional lattice.*

$$\mathcal{P}' = \mathbb{Z}^n \cap \mathcal{P}$$

Static Affine Nested Loop Program

Let us start with the most generic definition for the PM, as it provides the conditions for given a program to be described in the PM.

Definition 12. Static Affine Nested Loop Program (SANLP) *[138]. A SANLP consists of a set of statements and function calls, each possibly enclosed in loops and/or guarded by conditions. The loops do not have to be perfectly nested. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses have to be affine functions of enclosing loop iterators and static parameters. The parameters are symbolic constants, so their values can not change during the execution of the program. Data communication between function calls must be explicit.*

Static Control Parts

The next definition that comes after SANLP, moving to a finer granularity, is the one of Static Control Parts (SCoPs). A SCoP is a subclass of general loop nests that can be represented in the polyhedral model [38].

Definition 13. Static Control Part. *A SCoP is a maximal set of consecutive instructions such that:*

- *the control structures are only* for *loops or* if *conditionals*
- *loop bounds and conditionals are affine functions of the surrounding loop iterators and the global parameters (values unknown at compilation time, but constant).*

Even if the definition of SCoPs may seem restrictive, a pre-processing stage can extend its applicability.

As said, SCoPs are a set of statements. A *polyhedral statement* is the atomic dowel of polyhedral representation, and can be defined as:

Definition 14. Polyhedral Statement. *A polyhedral statement is a program instruction that:*

- *is not an if conditional statement with an affine condition*
- *is not a for loop statement with affine loop bounds*
- *has only affine subscript expressions for array accesses*
- *does not generate control-flow effects*

The resulting statements in the polyhedral representation may differ from those in the input source code, because the compiler may change the internal representation.

Iteration Domain

Iteration Domains capture the dynamic instances of all statements – *i.e.* all possible values of surrounding loop iterators – through a set of affine inequalities. In order to get to the definition in a rigorous manner, let us first of all define what an *iteration vector* is:

Definition 15. Iteration Vector. *For a polyhedral statement, the iteration vector of a multi-level loop nest over a* m*-dimensional grid is a vector of iteration variables,* $\vec{i} = (i_0, i_1, ..., i_{m-1})^T$*, where* $i_0, ..., i_{m-1}$ *are the iteration variables from outermost to innermost loop.*

Starting from the iteration vector, the Iteration Domain can be defined as:

Definition 16. Iteration Domain [81]. *The Iteration Domain (ID)* $\mathcal{D} \subseteq \mathbb{Z}^m$ *is the set of iteration vectors of the loop nest, and is expressed by a set of linear inequalities* $\mathcal{D} = \{\vec{i} \mid \boldsymbol{P}\vec{i} \geqslant \vec{b}\}$

Each integral point inside this polyhedron corresponds to exactly one execution of a statement, and its coordinates in the domain matches the values of the loop iterators at the execution of this instance. This model let the compiler manipulate statement execution and iteration ordering at the most precise level.

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <n; ++j)
        if(i<=n-j+2)
            s[i]=...
```

$$\mathbb{D}_{S0} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ -1 & -1 & 1 & 2 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$

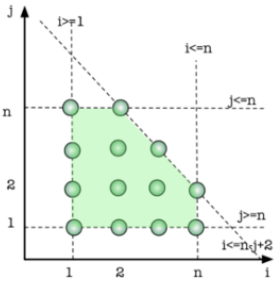


Table 4.2: *ID Example.*

Notice that, to model IDs whose size are known only symbolically at compile-time, parametric polyhedra are used.

Since the definitions of iteration vector and ID have just been introduced, the notion of lexicographic order can now be provided, as it will be useful to effectively model both *data dependencies* and *schedules*.

Definition 17. Lexicografic Order *[81]. Lexicographic order relation* $\succ_l$ *of two iteration vectors* $\vec{i}$ *and* $\vec{j}$ *is defined as:*

$$\vec{i} \succ_l \vec{j} \Leftrightarrow (i_0 > j_0) \vee (i_0 = j_0 \wedge i_1 > j_1) \vee (i_0 = j_0 \wedge i_1 = j_1 \wedge i_2 > j_2) \vee ... \\ \vee (i_0 = j_0 \wedge ... \wedge i_{m-2} = j_{m-2} \wedge i_{m-1} > j_{m-1})$$

Data dependencies

The modeling of data dependencies is crucial for the effectiveness of the PM, since not all program transformations preserve the semantics, and the semantic is automatically preserved if the dependencies are preserved. Here, some important definitions for data dependency analysis and representation are given.
Firstly, an essential definition to model the dependencies in the PM is the *subscript function*, as well as the notion of *image* and *preimage*.

Definition 18. Subscript Function [36]. *Given the set of array $\mathcal{A}_P$ of a program P, a reference to an array $B \in \mathcal{A}_P$ in a statement $S \in \mathcal{S}_P$ is written $\langle B, f \rangle$, where f is the* subscript function. *If f is affine it can be written as $f(\vec{x}) = F\vec{x} + \vec{a}$ where F is the* subscript matrix, *$\vec{a}$ is a constant vector.*

```
for (i = 0; i <= n; ++i){
    s[i]=0;
    for (j = 0; j <n; ++j)
            s[i]=s[i]+a[i][j]*x[j];
}
```

$$f_s(\vec{x}_{S2}) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} \vec{x}_{S2} \\ n \\ 1 \end{pmatrix}$$

$$f_a(\vec{x}_{S2}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} \vec{x}_{S2} \\ n \\ 1 \end{pmatrix}$$

$$f_x(\vec{x}_{S2}) = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} \vec{x}_{S2} \\ n \\ 1 \end{pmatrix}$$

Table 4.3: *Subscript Function Example. The three subscript functions are relative to the three array accesses for s, a and x.*

Definition 19. Image. *The image of a polyhedron $\mathcal{P} \in \mathbb{Z}^n$ by an affine function f: $\mathbb{Z}^n \to \mathbb{Z}^m$ is a $\mathbb{Z}$-polyhedron $\mathcal{P}'$:*

$$\mathcal{P}' = \{ f(\vec{x}) \in \mathbb{Z}^m | \vec{x} \in \mathcal{P} \}$$

Definition 20. Preimage. *The preimage of a polyhedron $\mathcal{P} \in \mathbb{Z}^n$ by an affine function f:$\mathbb{Z}^n \to \mathbb{Z}^m$ is a $\mathbb{Z}$-polyhedron $\mathcal{P}'$:*

$$\mathcal{P}' = \{ \vec{x} \in \mathbb{Z}^n | f(\vec{x}) \in \mathcal{P} \}$$

The image of a polyhedron by an affine invertible function is a $\mathbb{Z}$-polyhedron. The image of a polyhedron by a subscript function f_A in an ID $\mathcal{D}_S$ is the set of cell of A accessed from the statement S.

Thanks to those notion, the *data domain* (or *data space*) of a given array reference can be easily modeled. In fact, it is enough to compute the image of the ID of the statement by the reference subscript function.

Within the context of PM dependencies analysis, there is another important definition that must be provided, as it can be useful to check for the legality of a given transformation, but it can be employed for a whole lot of other purposes. This definition is the so called *data distance vector*, which comes together with the definitions of *lexicographically non-negative distance vector* and as an extension the *legality* condition for a given distance vector.

Definition 21. Data Distance Vector. *Consider two subscript functions* f_A^R *and* f_A^S *to the same array* A *of dimension* n*. Let* ν *and* σ *be two iterations of the innermost loop. The data distance vector is defined as an n-dimensional vector:*

$$\delta(\nu,\sigma)_{f_A^R f_A^S} = f_A^R(\nu) - f_A^S(\sigma)$$

Definition 22. Lexicographically non-negative Distance Vector. *A distance vector* ν *is lexicographically non-negative when the left-most entry in* ν *is positive or all elements of* ν *are zero.*

Definition 23. Legal Distance Vector. *A distance vector is legal when it is lexicographically non-negative (assuming that indices increase).*

In order to easily define the notion of *polyhedral dependency*, there is first the need to provide some introductory definitions, the first being the *Bernstein conditions*.

Definition 24. Bernstein Conditions [41]. *Given two references, there exists a dependency between them if the three following conditions hold:*

- *they reference the same memory location;*
- *one of this access is a write;*
- *the two associated statements are executed;*

Let us consider two statement *instances*, S_0, S_1, with S_0 occurring before S_1, there are three categories of dependencies that can be identified [105]:

- **Read After Write (RAW)**, S_1 reads what is written by S_0. If the dependency is not respected, S_1 incorrectly gets the old value.
- **Write After Read (WAR)**, S_1 write a destination after reading from S_0. If the dependency is not respected, S_0 incorrectly gets the new value.

- **Write After Write (WAW)**, S_1 write to a memory location after S_0. If the dependency is not respected, the writes end up being performed in the wrong order, leaving the value written by S_0 rather than the value written by S_1 in the destination.

As already stated, to preserve the semantic of the program, *instances* containing dependent references should not be executed in a different order. [154] classifies the dependency relation into three kinds:

- **Uniform dependencies**: the distance between dependent iteration remains constant
- **Non-Uniform dependencies**: during the execution the distance between dependent iterations varies
- **Parametric dependencies**: the distance between two dependent relations is expressed regarding to at least one parameter

Finally, let us define when two statements are said to be *in dependence* in the PM, leveraging the previously given definitions:

Definition 25. Dependency of statement instances. *A statement* S *depends on a statement* R *(*$R \rightarrow S$*), if there exists an operation* $S(\vec{x}_S)$ *and* $R(\vec{x}_R)$ *and a memory location* m *such that:*

- $S(\vec{x}_S)$ *and* $R(\vec{x}_R)$ *refer to the same memory location* m*, and at least one of them writes to that location*
- x_R *and* x_S *belong to the ID of* R *and* S
- *in the original sequential order,* $S(\vec{x}_S)$ *is executed after* $R(\vec{x}_R)$*.*

To effectively model dependencies between statements, a Data Dependency Graph can be employed.

Definition 26. Data Dependency Graph. *A Data Dependency Graph (DDG)* $G = (V, E)$ *is a directed multi-graph with each vertex representing a statement. An edge* $e \in E$*, from* R *to* S *represents a dependency between the source and target, due to a conflict access in* R *and* S*.*

Another useful representation in polyhedral theory is the *dependence polyhedron*, used in combination with the DDG. The dependency polyhedron provides the relation between the instances of the statements S and R. It is possible to obtain this kind of information

because there exists an affine relation between the iterations and the accessed data for regular programs, that can be obtained thanks to the previously defined subscript function. Before providing the definition of the dependence polyhedron, there is first the need to introduce the involved elements. First of all, the ID (being a set of affine inequalities) of S and R can be described as $A_S\vec{x}_S + c_S \geqslant 0$, and $A_R\vec{x}_R + c_R \geqslant 0$, where $\vec{x}_S$ and $\vec{x}_R$ are the iteration vectors of S and R. A dependence between S and R means that they refer to the same memory location, which implies that the two subscript functions are equal, hence $F_S\vec{x}_S + a_S = F_R\vec{x}_R + a_R$ (both expressed as in definition 18). There is also a precedence order between S and R, at the given *dependence level*, *i.e.* the common loop depth l in which the dependency takes place. For each dependence level l, the precedence constraints are:

- the equality of the loop index variables at any depth lesser to l:
$$x_{R,i} = x_{S,i} \ \forall i < l$$
- S is executed after R at the common depth l:
$$x_{R,l} < x_{S,l}$$

If S and R does not share any loop, there is no additional constraint and the dependence only exist if S is syntactically after R. These constraints can be expressed using linear inequalities, *i.e.* $P_{l,S}\vec{x}_S - P_{l,R}\vec{x}_R + b \geqslant 0$.

Definition 27. Dependence Polyhedron. *The dependence polyhedron* $\mathcal{D}_{R,S,f_R,f_S,l}$ *for R→S at a given level* l *and for a given pair of references* f_R, f_S *is described as:*

$$\mathcal{D}_{R,S,f_R,f_S,l} : D_{R,S}\begin{pmatrix}\vec{x}_R\\ \vec{x}_S\end{pmatrix} + \vec{d}_{R,S} = \begin{bmatrix}F_R & -F_S\\ A_R & 0\\ 0 & A_S\\ P_R & -P_S\end{bmatrix}\begin{pmatrix}\vec{x}_R\\ \vec{x}_S\end{pmatrix} + \begin{pmatrix}a_R - a_S\\ c_R\\ c_S\\ b\end{pmatrix} \begin{matrix}=0\\ \geqslant 0\end{matrix}$$

Given all the definitions above, the **Polyhedral Model** can be finally defined:

Definition 28. Polyhedral Model *[202]. The* polyhedral model *of a sequential program consists of a list of statements represented by:*

- *an identifier;*

```
for (i = 1; i <= 3; ++i){
    s[i] = 0;
    for (j = 1; j <=3; ++j)
            s[i]= s[i] + 1;
}
```

$$\mathcal{D}_{S1\delta S2} = \left[\begin{array}{cccc|c} 1 & -1 & 0 & 0 & \\ \hline 1 & 0 & 0 & -1 & \\ -1 & 0 & 0 & 3 & \\ 0 & 1 & 0 & -1 & \\ 0 & -1 & 0 & 3 & \\ 0 & 0 & 1 & -1 & \\ 0 & 0 & -1 & 3 & \end{array}\right] \cdot \begin{pmatrix} i_{S1} \\ i_{S2} \\ j_{S2} \\ 1 \end{pmatrix} \begin{matrix} = 0 \\ \hline \geq 0 \end{matrix}$$

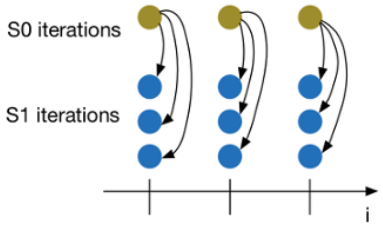


Table 4.4: *An example of a dependence polyhedron. In this example the polyhedron over the iteration vectors (one for the first statement, two for the second) and the scalar part are condensed in a single matrix (notice the 1 after the three iteration vectors). On the right there is a visual representation of the dependencies among the instances of the two statements.*

- *a dimension* $\mathrm{d_i}$*;*
- *an ID;*
- *a list af accesses;*
- *a location;*

A subscript function and a type (read or write) are associated to each array.

Schedules

The ID does not describe the order in which each statement instance has to be executed with respect to other instances. A *scheduling* function specifies a virtual timestamp for each instance of a corresponding statement, providing an order relation between statement instances. Hence, statement instances will be executed according to the increasing order of the timestamp. If two instances have the same timestamp can run in parallel.

```
for (i = 0; i < ni; i++)
    for (j = 0; j < ni; j++)
        if(i <= n-j+2)
            A[2*i+1][i-j+n] =0;
```

$$\theta_{S_1}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Identity Schedule

Table 4.5: *A simple schedule example. In this picture, the statement on the left has an* identity schedule*, as the statement instances are trivially the points* (i, j) *within the statement ID.*

Definition 29. Affine Schedule [153]. *Given a statement* S*, a* p-dimensional affine schedule Θ_S *is an affine form on the outer loop iterators* $\vec{x}_S$ *and the global parameters* $\vec{n}$*.*

$$\Theta_S(\vec{x}_S) = T_S \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}, T_S \in \mathbb{K}^{p \times \dim(\vec{x}_S)+\dim(\vec{n})+1}$$

A schedule assigns a timestamp to each executed instance of a statement. A schedule can be:

- **One-dimensional**, if T is a vector;
- **Multidimensional**, if T is a matrix.

A one-dimensional schedule express the program as a single sequential loop, while a multidimensional schedule expresses the program as one or more nested sequential loops [157].

There are however *schedules* which by construction are *not legal*, *i.e.* they enforce an execution order which violates the dependencies. The following definitions are essential to model this condition in the PM.

Definition 30. Precedence Condition. *Given* Θ_R *a schedule for the instance of* R*,* Θ_S *a schedule for the instances of* S*.* Θ_R *and* Θ_S *are legal schedules if* $\forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S,f_R,f_S.1}$ *(i.e. for each instance of* R *and* S *in dependence, as specified in the corresponding dependence polyhedron* $\mathcal{D}_{R,S,f_R,f_S.1}$*):*

$$\Theta_R(\vec{x}_R) \prec \Theta_S(\vec{x}_S)$$

Definition 31. Legal Shedule. *A* schedule Θ*, is* legal *if the* precedence condition *holds.*

Lemma 1. Affine form of Farkas Lemma. *Let* $\mathcal{D}$ *be a nonempty polyhedron defined by* $A\vec{x} + \vec{b} \geqslant \vec{0}$*. Then any affine function* $f(\vec{x})$ *is non-negative everywhere in* $\mathcal{D}$ *iff it is a positive affine combination:*

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}^T(A\vec{x} + \vec{b}) \text{ , with } \lambda_0 \geqslant 0 \text{ and } \vec{\lambda} \geqslant \vec{0}$$

λ_0 *and* $\vec{\lambda}^T$ *are called the Farkas multipliers.*

Tha Farkas lemma allows to translate the *precedence constraints* into an affine equivalent, *i.e.* an affine function. In order to satisfy the dependency $R \rightarrow S$ (definition 25), a schedule must satisfy [157]:

$$\Theta_R(\vec{x}_R) < \Theta_S(\vec{x}_S)$$

for each point of the dependence polyhedron $\mathcal{D}_{R,S_{f_R f_S l}}$. Hence:

$$\Delta_{R,S} = \Theta_S(\vec{x}_S) - \Theta_R(\vec{x}_R) - 1$$

must be non-negative everywhere in $\mathcal{D}_{R,S_{f_R f_S l}}$:

$$\Delta_{R,S} \geqslant 0$$

The set of legal schedules satisfying the dependency $R \rightarrow S$ is given by the relation:

$$\Delta_{R,S} = \lambda_0 + \vec{\lambda}^T \left(D_{R,S} \begin{pmatrix} \vec{x}_R \\ \vec{x}_S \end{pmatrix} + \vec{d}_{R,S} \right) \geqslant 0$$

where $D_{R,S}$ is the constraint matrix representing the dependence polyhedron $\mathcal{D}_{R,S_{f_R f_S l}}$ over $\vec{x}_R$ and $\vec{x}_S$, and $\vec{d}_{R,S}$ is the scalar part of these constraints, as described in definition 27.

4.1.3 Polyhedral Transformations

So far, the PM has been described, providing the mathematical toolset which allows to design sophisticated optimization heuristics by combining analysis power, transformation expressiveness and flexibility. In this section instead, the *framework* built on top of the PM is illustrated, in all of its phases: *analysis/representation*, *transformations* and as last step *code generation*.

Static Control Parts Extraction

The first task is obviously SCoPs extraction, as it allows the subsequent manipulations done in the successive phases. Briefly, it can be summarized by the following steps [37]:

1. **Information Gathering:** it consists of traversing the syntax tree of a given function, storing during this sweep *loop counters*, *bounds and strides*, *conditionals*, *array references*, and *parameters*.

2. **Affine Loops Recognition:** Once the collecting phase is done, identified loops are inspected in order to select the *static control* ones. First of all, bounds expressions are checked in order to extract only those with *affine* conditions. Then, conditionals are also checked to further refine the extraction, since they must be affine expressions of parameters and loop counters. Finally, only array references whose *subscript function* is also an affine expression of parameters and loop counters are selected.

3. **SCoPs Building:** In this phase the syntax tree is traversed once again, but this time aided by the previous extracted informa-

tion, and only for the part containing the loops remained after the aforementioned refinements, in order to build the set of SCoPs. First, a new SCoP is created; then, for each static operational or control node in the loop body:

- if it is a loop, this loop is added to the SCoP;
- if it is a conditional, then it is added with its branches to the SCoP;
- if it is not a conditional or a loop node, then it is added to the SCoP;
- otherwise, close the current SCoP and create a new one;
- drop the current SCoP if it eventually does not contain any loop

4. **Global Parameters Identification:** Finally, for each identified SCoP, iterate over loop bounds, conditionals and array references to collect global parameters.

Data Dependency Analysis

Once a function has been translated into the corresponding set of SCoPs, then data dependency analysis takes place. The objective of this phase is to compute the set of statement *instances* which are in dependence relation. Even though different approaches to this task have been proposed through the years, such as for instance the Omega Test [160], the widely accepted technique is that of *Data Flow Analysis* proposed by Feautrier in [80]. Starting from this work, the state-of-art-technique aims at building a Polyhedral Dependency Graph (PDG), consisting of a DDG in which, according to definition 26, nodes are the statements and edges are dependencies between them, and, for each edge, a corresponding dependence polyhedron, described in definition27. The procedure to build the DDG can be characterized by Algorithm 2

Then, by traversing the obtained DDG, the dependence polyhedra are built, as shown in Algorithm 3.

Note that the two operations can also be done concurrently, since the dependence polyhedra can be constructed right after each discovery of a new dependency (edge), resulting in a single algorithm. It must be also noticed that, whenever some types of dependencies are not needed, those dependencies can be simply not checked. In

Algorithm 2: DDG Construction

```
Create a graph in which every node is a statement
for all pair of nodes R, S do
  for all array references f_R, f_S do
    if f_R and f_S are on the same array then
      Compute the set Z of RAW, W of WAR, X of WAW dependencies
      if R ≠ 0 or W ≠ 0 or X ≠ 0 then
        Add an edge between node i and j
        Mark the edge with the array reference
        Mark the edge with the corresponding dependency type
      end if
    end if
  end for
end for
```

Algorithm 3: Dependence Polyhedra Construction

```
for all pair of nodes R, S do
  for all edge between those nodes e_{R,S} do
    if R and S does not share any loop then
      min_depth ← 0
    else
      min_depth ← 1
    end if
    for all level l from min_depth to number_of_common_loops do
      Build the Dependence Polyhedron D_{R,S,f_R,f_S.l}
    end for
  end for
end for
```

the case in which *data reuse* is the major concern, then also Read After Read (RAR) dependencies can be checked [46], although they don't actually belong to the canonical data dependencies categorization. Furthermore, redundant edges between nodes can be condensed obtaining what is called a Polyhedral Reduced Dependency Graph (PRDG).

Program Transformations

In a nutshell, the goal of a transformation is to modify the original execution order of the operations, *i.e.* the original *schedule*. At this point, OR comes into play, since transformations are always done targeting a specific optimization (or even more than one, in some cases) such as *latency*, *parallelism*, *data reuse*, and so on.

Obviously, in order not to alter the program's semantic, *i.e.* so as not to impair the correctness, a *legal* schedule must be found, *i.e.* the schedule which optimizes the given objective function must be selected within the *legal transformation space* [156, 157]. Hence, finding a good scheduling algorithm is basically a two-step approach [154]: the first consisting of finding the solution set of all legal affine schedules, the second consisting of finding an Integer Linear Programming (ILP) formulation for the objective function. After those two steps, an ILP solver can be used to find the optimal legal schedule.

The loop transformations achievable thanks to the PM are quite a few. Below, an overview of them is provided, and for some of them, the description comes along with a simple example.

Loop Reversal It basically reverses the order in which values are assigned to the index variable, changing the direction in which the loop traverses its iteration range. This kind of transformation can help to give space to further optimizations, previously not possible.

```
for (i = 1; i < ni; i++)
    for (j = 1; j < nj; j++)
        A[i][j] += A[i-1][j] + 1;
```

Listing 4.1: *Before*

```
for (i = 1; i < ni; i++)
    for (j = nj - 1; j >= 1; j--)
        A[i][j] += A[i-1][j] + 1;
```

Listing 4.2: *After*

Loop Interchange Also known as *loop permutation*, it consists of exchanging the position of two loops in a loop nest. It is mainly used

to improve cache effectiveness, modifying the behavior of accesses to arrays. Also, it can be used to control the granularity of the work in nested loops, interchanging for instance a parallel loop with a non parallel one, thus modifying the amount of work per parallel instance.

```
for (i = 1; i < ni; i++)
    for (j = 1; j < nj; j++)
        B[i] += A[i][j];
```

Listing 4.3: *Before*

```
for (j = 1; j < nj; j++)
    for (i = 1; i < ni; i++)
        B[i] += A[i][j];
```

Listing 4.4: *After*

This technique is however only legal if the distance vectors of the loop nest remains lexicographically positive after the interchange.

Loop Shifting It is a technique where operations inside a loop body are reordered. Obviously, it cannot be done whenever this reordering alters the dependencies. This transformation is sometimes referred also as *loop restructuring*.

Loop Fusion It consists of combining two loop bodies, and is also knows as *jamming*. The application of this transformation is safe only if no forward dependency between the two fused loops become a backward loop carried dependency. It is used in order to enhance data reuse, reduce loop overhead or eliminate synchronization between parallel loops.

```
for (i = 1; i < ni; i++)
    A[i] = B[i];
for (i = 1; i < ni; i++)
    C[i] = B[i] * A[i];
```

Listing 4.5: *Before*

```
for (i = 1; i < ni; i++){
    A[i] = B[i];
    C[i] = B[i] * A[i];
}
```

Listing 4.6: *After*

Loop Distribution also called *fission*, this transformation is basically the inverse of loop fusion. It breaks a single loop into multiple loops, iterating over the same index range. It can be done only if splitting the loop body does not alter dependencies between iteration instances. Its application can enable other transformations, and also reduce resource requirements, as well as allow partial parallelization.

```
for (i = 1; i < ni; i++){
    A[i] = B[i];
    C[i] = B[i] * A[i];
}
```

Listing 4.7: *Before*

```
for (i = 1; i < ni; i++)
    A[i] = B[i];
for (i = 1; i < ni; i++)
    C[i] = B[i] * A[i];
```

Listing 4.8: *After*

Loop Peeling This transformation consists of extracting one iteration of a given loop. It is done essentially to enable other kind of optimizations

Index-set Splitting Similar to peeling, but in this case the index set of the loop is splitted, so instead of extracting a single iteration, now the iteration space is divided among different loop instances.

```
for (i = 1; i < ni; i++)
    C[i] = B[i] * A[i];
```

Listing 4.9: *Before*

```
for (i = 1; i < ni/2; i++)
    C[i] = B[i] * A[i];
for (i = ni/2; i < ni; i++)
    C[i] = B[i] * A[i];
```

Listing 4.10: *After*

Loop Skewing It takes a nested loop iterating over a multidimensional array, in which each iteration instance of the inner loop depends on previous iterations, and rearranges its array accesses so that the only dependencies are between iterations of the outer loop. Technically speaking, the transformation makes the bounds of the inner loop depend on the outer loop counter, enabling inner loop parallelization.

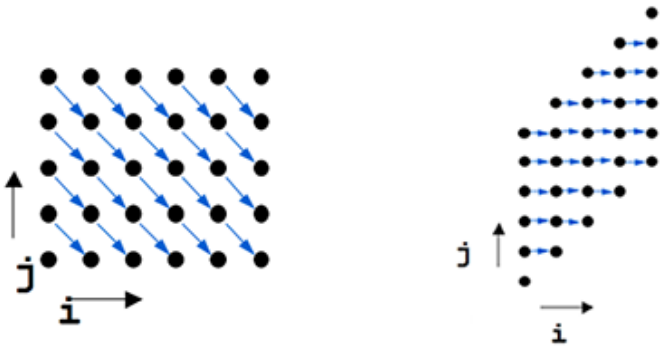
```
for (i = 1; i < ni; i++)
    for (j = 2; j < nj; j++)
  A[i,j] = A[i-1][j] + A[i][j-1];
```

Listing 4.11: *Before*

```
for (i = 1; i < ni; i++)
    for (j = ni + 2; j < i + ni; j++)
        A[i][j] = A[i-1][j] +
            A[i][j-1];
```

Listing 4.12: *After*

Tiling Sometimes known as *strip mine and interchange* or *loop blocking*, this transformation is used to enable coarse grain parallelism or



(a) *Original schedule* **(b)** *After skewing*

Table 4.6: *Loop Skewing Example. The ID is "skewed" to allow inner loop parallelization.*

enhance locality by making blocks whose data is sized to fit in the cache. What it does is partition the iteration space into tiles, whose size can be fixed or parametric [216]. A tile can be of three types:

- *Full Tile*: all points in the tile are valid iterations;
- *Partial Tile*: only a subset of the points are valid iterations;
- *Empty Tile*: no points are indeed valid iterations;

Obviously, an important task when doing *code generation* is to ensure that empty tiles are actually not visited, effectively reducing control overhead.

```
for(i = 0; i < N ; i = i + 1)
 for(j = 0; j < N ; j = j + 1)
  for(k = 0; k < N ; k = k + 1)
   C[i][j] = beta * C[i][j] + alpha *
       A[i][k] * B[k][j];
```

Listing 4.13: *Before*

```
for(jj = 0;jj < N;jj = jj + Bj)
 for(kk = 0; kk < N ; kk = kk + Bk )
  for(i = 0; i < N ; i = i + 1)
   for(j = jj;j < min(jj +Bj,N);j = j
       +1)
    for(k = kk; k < min(kk + Bk , N
        ); k = k + 1)
     C[i][j] = beta * C[i][j] + alpha
         * A[i][k] * B[k][j];
```

Listing 4.14: *After*

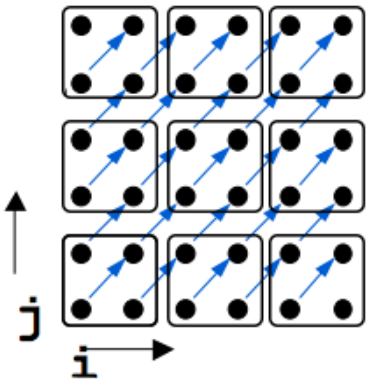


Table 4.7: *Loop Tiling Example. The ID is partitioned into the so called "tiles".*

Tiling Hyperplane Method Implemented in the state of art framework known as *PLuTo*, the tiling hyperplane method [46] is aimed at making the loop *tilable* (*i.e.* making tiling applicable) by computing a set of transformations, driven by an integer linear optimization formulation, done in order to minimize synchronizations and maximize locality. The computed transformations must ensure the following condition to be legal:

Lemma 2. Legality of tiling multiple domains with affine dependencies. *Let ϕ_{s_i} be a one-dimensional affine transform (i.e. schedule) for statement S_i.*
For $\{\phi_{s_1}, \phi_{s_2}, ..., \phi_{s_k}\}$ to be a legal (statement-wise) tiling hyperplane, the following should hold for each edge $e \in E$ of the PDG:

$$\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \geqslant 0, \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$$

where $\mathcal{P}_e$ is the dependence polyhedron associated to e.

Code Generation

Code generation is the last phase of program optimization through the PM. This is indeed a critical step in the polyhedral framework, simply because the effective optimization really depends on the target code quality. As the name suggests, it consists of regenerating the code in a given target language from the polyhedral representation obtained after the transformation step. This stage basically generates a *scanning code* [161] of the IDs of each statement, with the lexicographic order imposed by the current schedule. This scanning

code is an AST-based IR which is then quite easily translated into a target language, typically imperative, such as C.

In the early years of the PM, code generation was considered the bottleneck of the entire framework, due to the lack of scalability of the generation algorithms [153], mainly because of bad control management, which produced redundant conditions or complex loop bounds, as well as rapid code size explosion. This problem has been overtaken only recently, thanks to the work from Bastoul [34, 35], which proposed an extended version of the algorithm developed by Quilleré et al. [161]. The proposed technique from Quilleré et al., in which the essential part was a *recursive* generation of the scanning code (the Abstract Syntax Tree (AST)), was the first algorithm able to eliminate redundant control in the target code, but not able to deal with predicates and their impact on the control-flow, resulting easily in unacceptable code size. The later version from Bastoul was instead able to effectively reduce code size and processing time. Lately, Bastoul's work has been further improved [200], reaching the ability to scale up to thousands of statements.

4.2 Streaming Systems in FPGAs

Early years FPGAs have been primarily used to implement a small amount of glue logic between other chips, simply because they were not mature enough to handle complex computations and large problem sizes. However, recent trends shows that FPGAs are becoming increasingly powerful, more and more aligned with Application-Specific Integrated Circuits (ASICs) performance, but also comparable to other computing devices, thanks to an improved production process, a reduced power consumption, an increased speed, a larger amount of resources, in addition to an increasing possibility of on-the-fly re-configuration. This proves that FPGAs can now be considered a computing platform on their own, able to deliver very high performance even for complex problems [186].

It is however obvious that, due to their completely different architecture, FPGAs cannot be used as replacement of the other available computing devices. Instead, ad-hoc solutions must be found in order to effectively exploit their potentialities, while abstracting implementation details to facilitate scaling.

Streaming-based systems are a perfect example in this sense, as they embody precisely the distributed nature of the FPGAs. The

flexible granularity of those devices, in combination with memory elements distributed through the entire fabric, can easily deliver high quality results when used for such a purpose, granting high internal communication bandwidth while minimizing contention between elements.

Streaming-based architectures found their first applications in media processing [204], a type of computation well suited to be implemented in a streaming fashion, for the following reasons:

- The information, at least in an uncompresed form, is stored in multidimensional arrays;
- There is an enormous amount of information involved;
- Many of these algorithms do not need simultaneous access to the entire data array, as indeed processing usally operates on bounded regions (few frames, a single frame, or even a portion of a frame)
- The data access pattern is typically fixed.

However, due to the nature of certain regular computations, which enjoy the above properties as well, lately streaming-based systems have also been employed for a whole lot of other purposes, especially in the High Performance Computing (HPC) field.

Below, the working principles of a generic FPGA-based streaming architecture are explained.

4.2.1 Streaming Architectures

A stream-based processing system can be viewed as a Multiple Instruction Single Data (MISD) architecture [204], although the individual processing elements may themselves be SISD, SIMD, or MIMD in nature. It tends to be organized in a *systolic* structure, in which neighbours communicate directly through dedicated channels, implemented as FIFOs, and the computation is performed as the data streams flow through the corresponding units.

However, since storage capacity of FPGAs is relatively low with respect to the problem size of real applications, those architectures usually rely on external memory systems, employing specific logic demanded for communication with those systems, such as Direct Memory Accesses (DMAs).

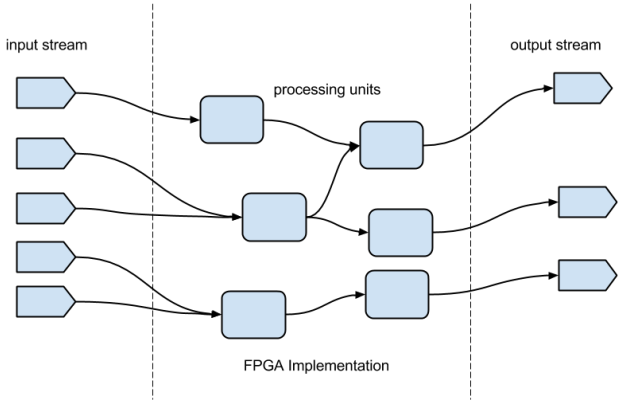


Table 4.8: *Streaming Computing: A General Picture.*

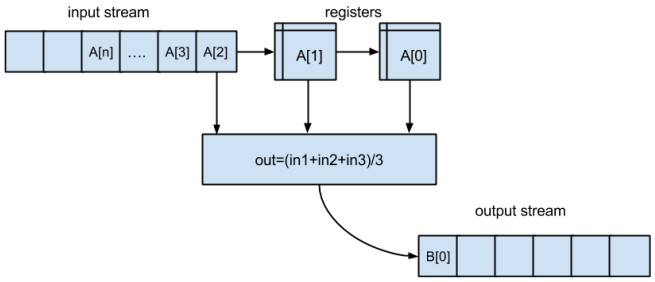


Table 4.9: *Generic Streaming Architecture.*

For these architectures, the memory interface is the key part of the entire system. In fact, in order to provide data at a sufficient rate, the input arrays are *linearized* into a mono-dimensional stream, and partitioned into smaller sub-blocks, following the array access patterns. This kind of *explicit management* of the memory, although it requires an additional effort with respect to traditional memory systems, avoids completely resource contention, allowing multiple concurrent accesses. Such an arrangment of the memory interface is able to deliver very large bandwidth towards the computational units, at a cost of increased design complexity.

In summary, when translating a problem specification into the corresponding streaming architecture, there are two major steps:

- For the computaional part, instructions are mapped into processing units
- Regarding memory, it requires explicit management, as it is first splitted following data access patterns, and then organized as a *chain* of FIFO buffers, in order to break the stream allowing multiple concurrent accesses.

This can be easily represented as a *graph*, with computational nodes and memory blocks linked together by *streams*, implemented as dedicated channels (*i.e.* FIFOs), as previously stated.

4.3 High Level Synthesis

A higher level of abstraction, beyond Register-Transfer Level (RTL), is increasingly important and unavoidable due to the growing of System-on-Chip (SoC) design complexity. The latest generation of HLS tools offers: different languages coverage, platform-based modeling and a domain-specific approach [64].
The abstraction level used by the early generation of commercial HLS systems was partially timed, and because of that they were not widely adopted, since neither languages nor the partially timed abstraction were well suited to model behavior at high level. The following generation provided synthesis of circuits starting from high level languages, *e.g.* C-code specifications. This, with other technical advances, enabled their industrial usage. Nowadays there is a growing demand for high-quality HLS solutions; more and more

functionality can be integrated on a single chip, but this involves increasing the number of design teams and design time. Lately they are constantly improving and the industry is now starting to adopt them into their design flow [1, 50, 60, 88, 132].

4.3.1 What is HLS?

High Level Synthesis is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior [65]. The synthesis starts from an high-level specification of the problem, where behaviour is decoupled from timing. The input specification language is analyzed, first *Resource Allocation* is done, that is the specification of how many and which type of operator and memory elements are required. Then the *Scheduling* assigns each operation to a time slot (clock cycle). During *Resource Binding*, operations and data element are bound to specific operators and memory elements. Also the interfaces are generated, consisting of data and control signal, between periphery and circuits. The result is an RTL design, which is in turn synthesized to the gate level by the use of a logic synthesis tool. An HLS tool is characterized according to different criteria:

- **Input language:** a designer would have the possibility to specify the algorithm in a high-level language rather than an hardware oriented language. It is obvious that some restriction must be applied on the high-level language, but they should not cause excessive difficulty in expressing a certain behaviour.
- **Ease of use:** a clear and complete documentation must be provided to flat the learning curve. Also a well designed graphic user interface (GUI) can simplify the design.
- **Data Type:** In hardware the primitive data type is a single bit. Support for complex data types is usually limited to integers, so additional data types ease the transition from algorithm to RTL.
- **Design Exploration:** the tools evaluate different architectures and choose the one that fits the design specifications.
- **Verification:** this phase can be speeded up if a tool generates testbench together with the design, and integrating the

source code (the reference) and the generated design into one testbench.

- **Metrics:** the RTL design generated must have the information about latency, the estimated clock rate and resource usage. An HLS tool can process different RTL designs exploiting Domain Space Exploration (DSE).

4.3.2 Advantages

The synthesis can be optimized taking into account performance, power, and cost requirements of a particular system. Design abstraction is one of the most effective methods for controlling complexity and improving design productivity. Adopting an HLS flow, fewer lines of code are written, this reduces mistakes and saves time. A RTL implementation has a fixed microarchitecture and protocol, while an HLS code can be retargeted to different technologies and requirements, so it can be reused in other designs. More and more accelerators are included in a System-on-Chip. HLS is particularly appropiate to build the architecture in support of this accelerators.

When targeting FPGAs, designers have even more advantages in adopting HLS:

- Modern FPGAs have many pre-fabricated Intellectual Property (IP) components embedded; HLS tools can apply a platform-based design methodology, taking into account these components.
- HLS significantly reduces the design time, or achieve quality of results comparable to hand-written RTL, putting the performance-power trade-off in the hands of the designers.
- Thanks to the recent advances in FPGAs, many HPC applications can be accelerated on a reconfigurable computing platform. The software developers do not write in RTL, so ithighly automated synthesis flow from C/C++ to FPGA is mandatory.

4.3.3 Evolution

As the design complexity of integrated circuits grows, it arises the need to generate circuit implementations from high-level behavioral specifications. The first HLS tools targeted ASICs design, and

is CMU-DA [205], developed at Carniege Mellon University in the 1970s, where the design is specified using an Instruction Set Processor Specification (ISPS) language, and then translated into an intermediate data-flow representation, before producing RTL. The tool included code-transformation techniques, hierarchical design and included a simulator of the original ISPS language.

During the subsequent years other tools were developed, most of them were academic projects. These tools typically decompose the synthesis process into steps, such as *register binding*, *scheduling*, *datapath allocation*. Different algorithms were developed to solve each phase. Until 2000 the tools often used custom languages for design specification, and because of the RTL synthesis tools were not mature, the HLS tools were not widely accepted.

Different reasons have influenced the adoption and guided the evolution of early HLS tools:

- They utilized an intermediate language as input, instead of a high-level language; this implied a learning curve for software/hardware developers. Even when the tools started to include C language, they did not accept more than a language, complicating the software/hardware co-design or simulation.
- The specification was tool-dependent so the produced implementation was unlikely to be portable.
- The HLS tools were not able to meet timing/power requirements in real life design, because the algorithms focused on reducing the number of functional units, and they did not take into account the IP blocks on a specific platform such as DSP and Block RAM (BRAM).
- The tools were born when the design complexity was acceptable to be handled without HLS. So there was not the necessity to spend time learning a new unproven design methodology.

A breakthrough was made when the tools focused on C-like languages to capture design intent. In this way the tools are more accessible to the system designer, and facilitate software/hardware co-design and co-verification. However, the C-based languages are criticized to be only suitable for describing sequential software that run on a Central Processing Unit (CPU). In particular C/C++ has the following limitations from the hardware point of view:

- does not include constructs to specify accuracy, timing, concurrency, synchronization etc.,
- has complex language constructs, such as recursion, that lead to difficulties in synthesis.

To fill the gap between C/C++ and HDL the tools have included: hardware-oriented language extensions, libraries (SystemC [17]), compiler directives and restrictions/interdiction of dynamic construct. Hardware and software co-simulation can be done without rewriting the code, if pragmas and directives are used. Doing so, s standard C/C++ compiler can compile the code bypassing the pragmas.

Many HLS tools nowadays target FPGA platforms; improvements made on these platforms, make them attractive for many applications. Some of the tools focus on a specific application domain, such as Digital Signal Processor (DSP) or floating-point scientific computing applications.

4.4 Application Domain

The class of problems I am targeting is that of *scientific workloads*. In fact, these algorithms can be easily written as:

- Static
- Pure
- Affine

imperatives codes.

I focus on this kind of workloads because all of the information needed are known at compile time. All the transformations on the source code can thus be done only analyzing the code statically. As most scientific workloads share these characteristics, we are able to analyze them more efficiently and, as we will see later, effectively and automatically parallelize the computation.

Now I describe how and when a code is static, pure and affine.

4.4.1 Staticness

Given a C code, it's possible todefine it *static* if:

- All loop bounds are known at compile time

- There are no data dependent conditional statements

Algorithm 4: Example of a static code

```
1: define M 10
2: define N 10

3: for i=1 to N do
4:    for j=i to M do
5:       if j <= 2 then
6:          b[j] = Func()
7:       end if
8:    end for
9: end for
```

4.4.2 Affinity

Given a code we define it *affine* if accesses to arrays happen using indices, constants or linear combinations of the indices of the enclosing loops. For example, Code 4 is also affine since data are also accessed linearly using j alone. An example of an affine but not static code is:

Algorithm 5: Example of an affine non static code

```
1: define M 10
2: define N 10

3: for i=1 to N do
4:    for j=i to M do
5:       if j <= a[i] then
6:          b[i*2][j+3*i] = Func()
7:       end if
8:    end for
9: end for
```

Note: in line 5 the *if-statement* depends on a data value, breaking the second condition for staticness. Each index in code 5 is a linear combination of enclosing indices and constants.

4.4.3 Pureness

Before specifying a condition for a *pure* code, it is useful to define what a *pure function* is.

Pure functions

A function is pure if:

- No read and write happens without the compiler knowing about it
- Result must not depend on hidden values (to the compiler) or any global state information
- It must not alter any input mutable parameter
- No global (i.e. shared) data

Pureness restricts code by not allowing to pass value by reference, in order not to share a global state.

Thusly, code is pure when all function calls are pure functions.

The following pseudo code shows an example of a pure code.

Algorithm 6: Example of a pure code

```
1: define M 10
2: define N 10

3: func foo()
4: a[]
5: for  i=1 to N do
6:    for  j=i to M do
7:       if j <= 2 then
8:          b[j] = Func(a[i])
9:       end if
10:   end for
11: end for
12: endfunc
```

The following code is not pure, since it accesses a global variable via reference.

Algorithm 7: Example of a non pure code

```
1: define M 10
2: define N 10
3: a[]

4: func foo()
5: for i=1 to N do
6:   for j=i to M do
7:     if j <= 2 then
8:       Func(&a[i])
9:     end if
10:  end for
11: end for
12: endfunc
```

Beauty of pure functions

Pure functions map well on parallel hardware as they don't require the implementation of a global memory to exchange data, a potentially critical bottleneck in most systems. Indeed, implementing global state on hardware requires the presence of some kind of hardware synchronization mechanism, wasting resources and introducing wait states for all the components that rely on that information. Pure code prevents by design these kind of side effects.

4.5 Iterative Stencil Loops

Appropriate exploitation of HPC is nowadays of paramount importance for many scientific and engineering applications, as the increasing computational power has allowed to push the limits of what can be modeled and simulated, widening dramatically the range of problems that can be addressed. However, architectural trends show that there is a growing gap between time for processors to perform arithmetic operations and time they take to communicate [110], a limit which is unacceptable for memory bound computations such as ISLs, an important part of solvers in this field. In this section focus is on what ISLs are, their properties and characteristics, and how they are currently treated in the state of the art.

4.5.1 Definition

The so called Iterative Stencil Loops are basically a class of iterative algorithms, whose features make them belonging to the class of SANLP (see section 4.1.2), which consists of the repeated updating of values associated with points on a multi-dimensional grid, usually 2- or 3-dimensional, modeled as an array, using weighted contributions from a subset of its neighbors in both time and space. The fixed pattern of neighbors is called *stencil*, and the function that uses those elements to update an array cell is called *transition function*. An ISL can be generically represented by the pseudocode of algorithm 8.

Algorithm 8: Generic ISL Algorithm

```
for t ⩽ TimeSteps do
  for all points p in matrix P do
    p ← f_transition(stencil(p))
  end for
end for
```

As previously stated, the number of algorithms that fall into this category is quite large, which is why is of great concern to efficiently implement them, even if this is not an easy task at all. Indeed, a lot of algorithms for scientific computing, such as [39, 172, 187], as well as image and video processing, such as [52, 90], belong to this class and can be generalized in the form of algorithm 8

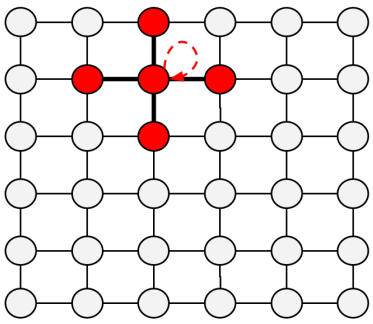


Table 4.10: *An illustration of a generic 5-point 2-Dimensional ISL.*

Formal Model

Formally speaking, an Iterative Stencil Loop can be defined as a *5-tuple* (I, S, S_0, s, T) [86] in which:

- $I = \prod_{i=1}^{k} [0, \ldots, n_i]$ is the index set. It defines the topology of the array.
- S is the set of states, one of which each cell may take on any given time-step.
- $S_0 \colon \mathbb{Z}^k \to S$ defines the initial state of the system at time 0.
- $s \in \prod_{i=1}^{l} \mathbb{Z}^k$ is the stencil itself and describes the actual shape of the neighborhood. There are l elements in the stencil.
- $T \colon S^l \to S$ is the transition function which is used to determine a cell's new state, depending on its neighbors.

Coefficient Types

As stated when ISLs has been defined, the contribution of the points of the stencil is usually *weighted* by some coefficients. The type of coefficients to which neighbours are weighted could lead to two scenarios:

- **Constant coefficients:** When the coefficient values are constant scalars, there is no need to read them repeatedly. They can be instead hard-coded into the stencil loop, resulting in a reduction of storage requirements and memory traffic. As intuition suggests, the case in which coefficients are constant is the ideal scenario, since stencil-related optimizations fully impact the resulting implementation.
- **Variable coefficients:** In this case the stencil weights can change during the execution, being different between time-steps or from one grid point to another. These weights are stored in separate grids streamed during the computation, which obviously causes an extra memory traffic. This requires special care as stencils are already memory-bound by themselves.

Boundary Conditions

Depending on the nature of the computation, two basic types of boundary conditions for the ISLs can be identified:

- **Constant Boundaries:** This scenario is the one in which boundaries are constant during the computation (figure 4.11a). This is the general case, in which they can be simply represented as a ghost zone of the stencil array, *i.e.* the one updated during the ISL computation. Furthermore, if it is the case in which these cells have all the same value, or at least they can be clustered into smaller sets than the entire number of ghost cells, they can be stored in fewer registers and referenced multiple times. This is obvioulsy a matter of implementation choiches, as it depends on the underlying computing architecture.
- **Periodic Boundaries:** In this case, the grid wraps around all its dimensions, an operation in mathematics called *compactification*. In the case of a two dimensional grid for instance, this means that the left boundary is adjacent to the right boundary, and the top boundary is adjacent to the bottom boundary. This kind of boundary type is often chosen to approximate large – or even infinite – systems. Obviously, this means that in this case also the boundary changes over time, as it is indeed updated during the computation, thus not allowing the optimizations available when dealing with constant boundaries (figure 4.11b).

(a) *An ISL with constant boundaries* **(b)** *An ISL with periodic boundaries*

Table 4.11: *ISLs boundary types.*

4.6 Long Term Vision

Trends described in 1.2.6, strongly hint to a future where heterogenous systems are the norm. More and more datacenters and

supercomputers are relying on heterogeneity to achieve faster and faster computing speed while maintaining the power consumption as low as possible.

Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts, but they allowed to quickly prototype components or build up circuits when ASIC production would be too expensive. Nowadays, thanks to technology advancements, FPGAs can realistically be seen as the next core heterogenous components in (near) future supercomputing. Right now, researchers are porting algorithms on this platform to achieve better throughput at lower power consumption than their GPUs counterpart [84, 89, 133, 139, 189, 199].

In a world where energy is an ever scarcer resource, I will rely more and more on this technology to achieve better power efficiency. What is restraining the use of FPGA is the higher learning curve and very complex design tools, compared to CPU and GPUs.

But, as the *green-scientific* becomes the hot topic, given the trends, FPGAs will implement more and more scientific algorithms, for improved power efficiency.

4.6.1 Main Characteristics and Implementation Challenges

When it comes to implement stencil computations, there are at least two important characteristics of those algorithms that must be taken into account, since they cause some cumbersome implementation challenges.

Memory Boundedness

The main difficulties that arise when implementing ISLs are due to the fact that the performance is bound by the memory transfers, mainly because of architectural limitations – memory is intrinsically slower than the computational units – but also due to the nature of these algorithms as they require multiple constant accesses to the stencil array.

On CPUs based platforms for example, the matrices on which the computation is performed, are much larger than the capacity of the available data cache [69, 110], causing continuous misses and resulting in penalties which inevitably slow down the execution.

Regarding FPGAs, the limited amount of memory resources can even lead to infeasibility for problems on large grids, not to mention

port contention on BRAM [59], which is by the way a major concern not only for ISLs, but for basically every FPGA implementation.

Furthermore, for ISLs *in general* there is always a bandwidth problem: in fact, memory slowness can cause the computation to stall if there is not enough data ready for arithmetic units [207], lowering performance with repsect to theoretical peak on every device, as the aforementioned, and including also General Purpose Graphic Processing Units (GPGPUs) [180].

Spatial dependence between grid points

Another important aspect to deal with, is the eventual presence of true data dependencies between updated points of the grid in the same time frame. Trivially, this yields the following two distinct cases.

- **Dependency-free points** This first scenario is the one in which there is absolutely *no* dependency between points of the grid, which implies that every point can be independently computed from each other. It basically means that updating of points is trivially parallelizable, giving space to a whole lot of optimizations, but due to the nature of stencil computations, this also comes with an important drawback, caused by the temporal dependency between different time-steps. In fact, when parallelizing, this dependencies require communication and synchronization for which non negligible overhead may incur, obtaining significantly lower performance than in theory.

  The Jacobi iterative method [172, 187], is an example of such a type of algorithm.

- **Dependent points** When the neighboring elements used in the stencil come also from the same time frame, *i.e.* the data used for an update comes from a computation made within the same time-step, this can lead to also spatial dependencies between points, enforcing an order of execution even in the same time-step. This sequential ordering implies that no – or at least not trivial – parallelization optimization can be made. The result is that, in this case, improvements are even harder to achieve than in the first scenario.

  The Gauss-Seidel method [172] is a perfect example of this category. A parallel version of this method has been however de-

veloped, namely *red-black* Gauss-Seidel [118], but it requires a specific traversal of the grid which by the way makes useless any kind of cache optimization, as switching from one set of points, *i.e.* color, from another, cause cache misses that, especially for large problems, are the dominating factor which negatively impact on performance [207].

4.6.2 State of the Art

The implementation challenges discussed so far in section 4.6.1, that arise when dealing with ISLs, have created an entire research branch focused only on optimizing stencil computations. The resulting extensive study has led to a wide range of different optimizations. Here, an overview of them is provided.

Tiling Based Optimizations

The first category is the one in which *tiling*, also known as *blocking* (see section 4.1.3), is employed to effectively improve performance by enhancing data locality and exposing parallelism. This technique has been exploited in a number of different ways, and performed in both spatial – when possible – and temporal dimension, resulting in a variety of classes [163], which are shown below.

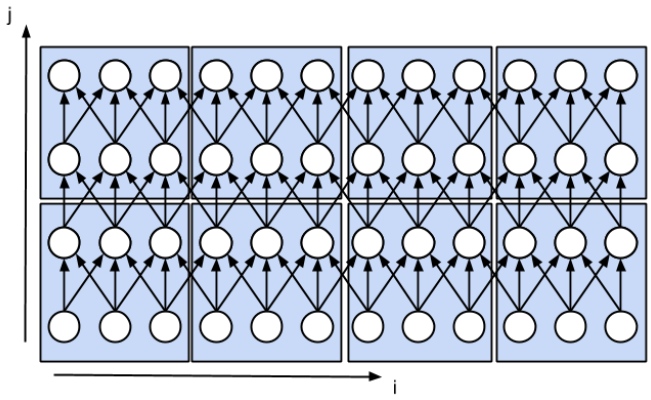


Table 4.12: *Single Iteration Tiling.*

Single iteration tiling This first type of tiling is the most trivial one, as it consists of applying conventional loop blocking to improve

cache reuse. In this case, a single time frame (*i.e.* a single iteration) is partitioned into smaller blocks, allowing points that are close in space to remain in cache when used, thus allowing to update them toghether, improving locality [110]. This technique has been also exploited to distribute the computation to multiple Processing Elements (PEs), in order to parallelize points computation within a single iteration [92], also leveraging specific Application Programming Interfaces (APIs) such as OpenMP [68]. However, tiling across multiple PEs reaches far from optimal results since stencils along the boundary of a tile require values that were previously computed by other PEs, resulting in an increasing need of communication and synchronization between tiles, proportional to the number of them. An effective technique to overcome this issue is the one known as *ghost zone optimization* or *overlapped tiling* [107, 117, 134], which basically consists in the enlargement of the tiles with ghost zones, *i.e.* the overlapping regions between tiles, replicating some computations but nevertheless reducing communication and synchronization. Although it may seem that applying this technique can always lead to better performance, despite replication, it must be noticed that an improper selection of the ghost zone size may result in even worse performance with respect to no optimization at all.

As last consideration, when dealing with ISLs which have also spacial dependencies between grid points, this type of optimization is not applicable, at least not for parallelization purposes, and performance are usually not satisfying also regarding cache optimizations [207].

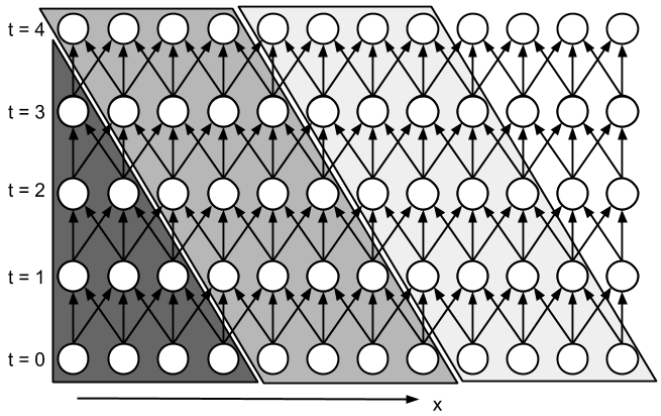


Table 4.13: *Time Skewing.*

Time skewing In this scheme of tiling, multiple iterations are *collectively* partitioned into blocks, so the essential difference between this strategy and single-iteration tiling, is that in this case multiple iterations are included as part of each tile. The reason beyond the application of such a strategy is to use also the temporal locality, and thus to increase the overall data reuse. However, in order to make tiling legal, loop skewing (see section 4.1.3) along the time dimension is required. In fact, due to the fact that points update is performed in both spatial and temporal dimensions in each block, they must shift their collection of points backward on the time dimension to respect temporal dependencies induced by the ISL, *i.e.* transform dependency distances into non-negative values [168], resulting in a loss of inter-tile concurrency, because of the fact that the skewing introduces inter-tile dependencies in the spacial direction. As it may seem that this variant could always deliver better performance than the simple single-iteration tiling, it actually really depends on a careful selection of the skewing factor [163], as well as on the form of the tile [168, 188], which can be a major concern especially on FPGAs [219]. With respect to the previously mentioned strategy, time skewing can provide better cache hit rates and effectively reduce processor idle time caused by ISL's memory boundedness [212].

As for the previous tiling strategy, even in this case block distribution among different PEs is possible [21], but likewise single-iteration tiling, it requires explicit synchronization between them, since a block must wait for its neighbors to complete in order to have enough data to start. As a consequence of this needed scheme of synchronization, rather than a purely parallel execution, in this case blocks are executed in a pipeline fashion.

A possible solution to the necessity of time skewing when tiling along multiple iterations is proposed in [137], where *code transformation* is performed with the aim of fusing the stencil loops, in order to reduce the number of reads and writes, and increase instead the computational intensity. This is an effective solution to overcome the memory boundedness of ISLs, since usually the computational part of those algorithms is a fraction of the entire execution time, and enlarging it with a corresponding reduction of memory traffic can exploit the computational power of modern architectures. A very similar technique has been developed in [54], in which a domain-specific compiler is proposed, namely *Caracal*, able to perform un-

rolling of the time loop and fuse accordingly the stencils, with the same effects as of [137].

Wavefront parallelization This strategy is somehow similar to the previous, but instead of pipelining the execution of time-skewed blocks, these blocks are scheduled collectively in a *wavefront* fashion [180,196,208]. In this case then, instead of requiring explicit synchronization, blocks are arranged in a way that on the time dimension the computation blocks are independent from each other, thus not requiring synchronization.

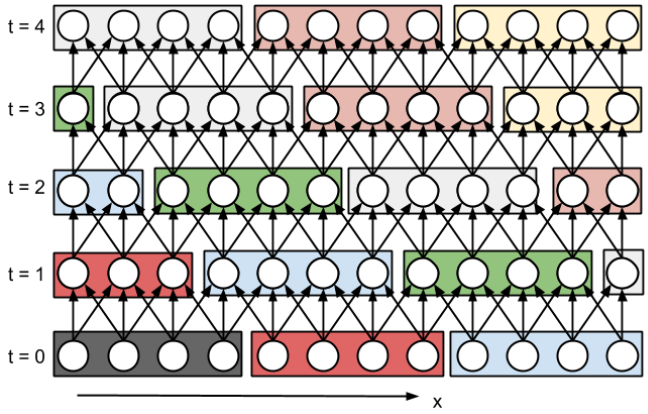


Table 4.14: *Wavefront Parallelization.*

Although in [163] this scheme has been explicitly defined as the one in which *multiple* blocks are *scheduled together*, this class can be easily extended to the case in which only *one* block implements a single iteration. In fact, these collections are executed in a pipeline fashion along the time dimension, coming with no need of synchronization. Indeed, this is exactly the kind of behavior exhibited when tiling is *only* applied on the time dimension [146,175,176], and by far this approach is the most promising one with respect to the previous two, as it has been proven to be scalable [146, 175, 176], and comes with no communication overhead.

In a sense, the work proposed in this thesis can be at least partially included in this class.

Domain Specific Language (DSL) Based Optimizations

Another important approach towards optimization, extensively used in literature, and which is becoming increasingly popular, consists of the exploitation of Domain Specific Languages (DSLs) and ad-hoc frameworks. As ed General Purpose Languages (GPLs) are the dominating software development tools in HPC, the lack of specialized features for narrow domains such as ISLs is a great limitation, since most of the times it does not allow to express a problem in a way which is easy to manipulate, making optimal implementation an hard task.

In this sense, Domain Specific Languages are certainly more powerful as they provide, the ability to define a problem within the specific application domain at a cost of losing broad applicability (although in some cases it could be still technically possible). In such a way some features and some specific semantics are explicitly expressed, which enables whole sorts of transformations, manipulations and optimizations simply not possible – or hardly achievable – using general purpose languages.

Currently, the ISL domain can count a number of available DSLs, each with its own peculiarities. For instance, PATUS [55] is able to achieve a high performance by auto-tuning, targeting different hardware architectures, while Pochoir [193] provides a C++ template library based on a divide-and-conquer skeleton which is then translated into Cilk [44], a C/C++ extension designed for multithreaded parallel computing. ExaStencils [122] employs a direct mathematical formulation (ExaSlang) of the problem, and through a series of steps of transformations, included a wide range of PM-based optimizations, generates target code in a specific language, which by now is C/C++, but in the future could be extended to other languages. DeLite [190] abstracts from Scala with the aim of making stencil programming easier, and uses metaprogramming to construct an IR of the problem and compile to a large number of languages, so that it can easily target heterogeneous hardware. In [217], a single mathematical formula is used to implement 3-D stencil codes on GPGPUs, via auto-tuning and automatic target code generation, and GPGPUs is also the target device of [107], in which low-level code is generated, starting from an abstract representation, by trading an increase in the computational workload for a decrease in the required global memory bandwidth. In [209] a single high-order function specified

in Haskell, and specifically in CλaSH [26], a functional HDL able to translate plain Haskell (with some restrictions) into synthesizable VHSIC Hardware Description Language (VHDL), is used in combination with a series of transformations to generate hardware accelerators.

As final consideration, although using DSLs can lead to good performance, as previously stated in HPC this is not the common practice at all, as GPLs are preferred due to their versatility and ease of use. This trend is not going to change, at least in the near future, and because of that, trying to achieve the best from General Purpose Languages is still an important but nevertheless challenging task.

Custom architectures

When designing custom hardware, FPGAs offers a high flexibility, and nevertheless can deliver sustained performance with high energy efficiency, often orders of magnitude better than other hardware platforms. Due to those interesting characteristics, FPGAs have been extensively used as target device for the optimizations aforementioned, but they express their real potential especially when designing custom microarchitectures. In fact, an increasing number of works are focusing on exploiting FPGAs to implement ISLs with the production of ad-hoc hardware, finely designed to efficiently leverage the regular structure of this class of algorithms, which allow complete compile-time analysis. In particular, this kind of approach has been proved to be especially useful to overcome the memory boundedness issue of stencil computations.

In [169] for instance, a generic tunable VHDL template has been proposed to parallelize 3-D stencil computations, which use, in favor of Partial Buffering (PB) where only the data needed by the current computation is stored to minimize memory consumption, the so called Full Buffering (FB) [126], a technique in which data is stored on the on-chip memory until all the computation relying on it has completed, showing that the increasing number of available resources in modern FPGAs has made the time ripe enough to allow to push the limits of what can be achieved on such a device.

In [59] the PM is employed to take advantage of the stencil access pattern and perform non-uniform memory partitioning in order to generate a custom microarchitecture, streaming oriented, which is proven to be optimal with respect to memory usage, since it allows

FB with the minimum number of reuse buffer banks and minimum buffer size. Although this architecture has been never really tested – it has been actually only simulated – to prove its validity, and the case in which the computation has as input other matrices than the stencil one is not covered, the ideas behind this work are still of great value, so that they have been used in this thesis as basis for the development of the proposed custom memory microarchitecture.

In [116] 2-D stencils are addressed using *ScalableCore*, a system composed of multiple small capacity FPGAs, connected in a 2-D-mesh. To efficiently exploit such an architecture, the stencil computation is tiled and each computational block is assigned to an FPGA, and in order to overcome the communication overhead introduced by tiling, the execution order is customized in each FPGA. The work demonstrate that a custom FPGA-based architecture can deliver power efficiency much higher than traditional computing devices.

In [182] a memory architecture is developed to implement symmetric 3-D stencils, *i.e.* of the form of $n \times (n+1) \times n$, which use First In First Out (FIFO) queues for both the input and output stream, one for each dimension, a *data engine* (also called front-end) which prefetches data, a *compute engine* (the back-end), which consists of multiple instances of the computation unit, and a *control engine* responsible for synchronizing the data flow in the whole architecture.

CHAPTER 5

On how to Explicitely Isolate Data and Computation

In this Chapter we introduce a novel technique to accelerate algorithms and codes that expose plenty of data level parallelism, a typical situation when analyzing High Performance Computing (HPC) codes. We assume that codes can be modeled as Static Affine Nested Loop Program (SANLP), an can be analyzed by means of Polyhedral Analysis (PA). When this assumptions are right, we demonstrate via the undermentioned methodology how generating multiple *tiled* hardware accelerators benefits the overall performance of the resulting system (both in terms of power efficiency and throughput).

5.1 Introduction

In the race to exascale computing, academic and industrial research is focused on dramatically improving the power efficiency of nowadays computing systems. In order to attain this goal, different

kinds of heterogeneous systems are being employed as a successful approach, as recent trends show.

Indeed, modern power-efficient supercomputers employ Graphic Processing Unit (GPU) as privileged components to attain dramatically improved overall power efficiency than Central Processing Units (CPUs)-only based systems, on specific workloads, notably those exposing plenty of data level parallelism, the key to extract performance from spatial architectures. While there is still plenty of room for further improvements, the nature of the underlying architectures of GPUs will eventually plague power-efficiency in a similar fashion to what is happening to CPUs on said workloads

Developing on these considerations, a consistent body of research elaborates on the benefits deriving from the employment of Reconfigurable Hardware (RH) as the core computational element in future heterogeneous system to attain the best power efficiency. However, traditional toughness associated with the development of hardware based reconfigurable accelerators, usually implemented on Field Programmable Gate Arrays (FPGAs), makes the transition and the further development of state of art, power efficient RH based systems very tough.

High Level Synthesis (HLS) plays a central role in further development of RH as core computational element of future heterogeneous systems, as it both dramatically reduces the burden on the application designer and drastically speeds up the exploration of different hardware design tradeoffs. However, a major limitation of modern HLS tools is that, while it is possible to quickly explore simple, well known optimizations such as loop unrolling and pipelining, obtaining high quality results, this is not true for most of the complex and powerful generic polyhedral affine transformations. This limits the ability to better explore area/power/communication trade offs, specifically in respect to data-parallel workload distribution.

Additionally, while the last decade saw an increasing effort put into the development of techniques aimed at improving the quality of the resulting hardware components, relatively less effort has been spent on System Level Design (SLD) aspects, leaving application designers of RH based heterogeneous systems the entire effort of planning how to distribute workload among heterogeneous computational elements. It is difficult to design an hardware accelerator with the right trade-off between computation/ communication/

parallelism/ memory efficiency.

In this work we address both those problems with a focus on amply data-parallel codes, and provide the following contributions:

- *first, we develop a novel HLS approach to using Polyhedral Model (PM) as a means to explicitly extract and isolate data and computation from affine codes in order to efficiently divide the workload among an arbitrary number of nodes, in the light of the current and foreseeable trend of adoption of reconfigurable hardware in the datacenter*
- *second, and towards energy proportional computing, we improve the current state of art in single core acceleration, as our methodology obtains near-linear speedup with the area at disposition to accelerate the given workload.*

The rest of the chapter is organized as follows: in Section 5.2 we give a thorough review of the state-of-the-art and related works. In Section 5.3 we detail the contribution of this work, and its limitations of applicability. The experimental results are provided in Section 5.4. The chapter is concluded in Section 5.5, along with hints about future work.

5.2 Related Work

Recent years have seen dramatic improvements in High Level Synthesis (HLS) tools; efficient translation of numeric [206], image processing algorithms [150], and stencil computations [219], where relevant application domains benefits from both research on compute and/or memory/communications network aspects.

Due to the well-known drawbacks HLS has (such as area overhead and slow downs with respect to manually crafted designs) last decades' studies focused on more formal approaches to *systematically* synthesize better circuits; one notable framework in this context is the PM and the associated code analysis technique, collectively called Polyhedral Analysis [12, 35, 36, 38, 125, 149, 153, 154, 158, 165, 201–203, 218, 219]. Under this representation it is possible to compute dependencies, find bounds, and reorder instructions in a completely automated manner relying on the same set of sound and comprehensive assumptions of PM.

The most relevant tools for doing PA are Polly [101], Polyhedral Compiler Collection (PoCC) [18] and PLuTO [46]. The first is

a popular LLVM plugin capable of applying PA on its intermediate representation right before invoking the backend (which could, for example, be LegUp [7, 50]). The second is a popular compilation suite supporting polyhedral analysis, with an HLS-oriented version called PolyOPT/HLS [123]. The last is a compilation suite capable of auto tuning the affine transformations made to the original code in order to achieve quasi-linear speedup with the number of cores by simultaneously applying code restructuring, improved data locality, and automatic OpenMP parallelization. Variants of the PA-based techniques already implemented in production compilers can be applied to HLS as well, with adequate measures to cope with the specific requirements of hardware generation [155, 218, 219] and memory architecture considerations [59, 62, 131, 159, 206].

As a hot topic, PM is the kernel of many current research directions. We cite two of them, relevant to this work.

The first one is that of the Daedalus framework, proposed in [143–145], where authors employ a tool – PNGen [203] – to derive a specialization of a Khan process network called Polyhedral Process Network (PPN). PM is employed to model the target code as a parallel network of (pure) processes exchanging data through bounded First In First Outs (FIFOs), where reads and writes occur in a (guaranteed) deadlock-free order inferred at compile time from C code. In the Daedalus framework the nodes are general purpose processors of a Network on Chip (NoC). Broadly speaking, Daedalus framework maps a PPN onto a Multi Processor System on Chip (MPSoC). It is worth to note that not every node of the network can run concurrently: this contrasts with our approach, in which we synthesize entire portions of parallel sections into isolated portions of code and corresponding data, while guaranteeing that those synthesized accelerators will actually run in parallel.

The other research direction was pointed out in [21, 72, 136, 159, 218, 218, 219]. In their works they propose an automated framework capable of extracting the polyhedral model and restructure the code to achieve better data access and reduce area utilization. Memory architectures for affine computation have been heavily explored in [59, 62, 131, 159, 206], and are orthogonal to this work. Indeed, they achieved better performance than simple HLS, but only limited to single core, where the inner HLS process prevents further SLD steps.

While many of the cited work elaborates on the techniques required to apply PA to HLS, no one directly elaborates on the role

of the PM to explicitly isolate data and corresponding code in order to separate this "unit of computation" from the rest of the accelerator. A new SLD technique is needed to allow large data parallel portions of computations to be arbitrarily split and isolated in order to distribute the resulting workload among multiple FPGAs, which is precisely the scope and the novelty of this work. Lastly, it is worth to note that most works related to memory architectures tailored around code modeled through the PM are orthogonal to this work, and are expected to be integrated as future work.

5.3 Data And Computation Isolation

As reported in [124,159], an effective way of improving throughput with loop pipelining is to eliminate loop-carried dependences. Indeed, affine programs usually expose an inner, embarrassingly data-parallel loop, possibly after adequate loop transformations that can be effectively represented using PM [218,219]. Even rich Domain Space Exploration (DSE) steps could be easily added by invoking tools like that presented in [12, 158] . While authors in [124, 159] focus on the acceleration of such loops by means of HLS pipelining directives, we further extend this concept by:

- identifying not only the innermost, but also higher-level loops that do not introduce loop-carried dependences, and,
- isolating data and code, by means of PM, in tiles that can be scheduled either in the same core or on (possibly remote) loosely-dependent accelerators.

We generate N sub-kernels capable to process an Nth of the input data produced by the statements of the analyzed loop. They are separated in two parts: the first is the Processing Unit (PU), i.e. the place where actual computation occurs. Hardware Description Language (HDL) is generated via HLS, to which is fed the code extracted after PA from the corresponding tile. The latter is enough local memory to contain the data required to produce the aforementioned Nth of the input data.

In order to isolate the instructions and the data they process, we rely on the tiling transformation [167]. We separate the workload by cutting the data domain into multiple *splits*, each of which is characterized by an amount of data to process and an the corresponding

instructions insisting on them. We tile by invoking parts of the PoCC toolchain [33,78].

However, we extended this transformation in the sense of data locality. In order to do so, we analyze the tiled code, and the corresponding access relations $\Gamma_{s,A}(\vec{\xi}_s)$. Then, we compute the amount of local memory required for *all the reads and writes to be local* (in FPGA terms, localized in Memory LUTs or BRAMs). We transform HLS code so that data required by the cores is sent through streaming interfaces to them, and such that the references occur – in HLS – in the local store. The local store of the generated cores is employed as a cache: glue logic is responsible for correctly treating the reads and writes.

Moreover, this restructuring of code and local stores enables, *by design*, the activation of some specific HLS optimizations, like data flow and pipelining, which are directives that are sometimes not available at all without proper code restructuring. As these optimizations are always available after memory reshaping, we enable them during core-generation as part of the methodology.

We refer to each of these sub-kernels as *computational unit*. For each isolated computational units, we compile a function in the HLS source code, with the additional required glue logic and the streaming interfaces that connect the computational units to the rest of the user logic. Part of that user logic requires that, before the next iteration starts, all the data processed by computational units be read, as a means for synchronization. The process and the resulting architecture is depicted in Figure 5.1. This separation in multiple computational units will introduce some communications costs, but as we will show in the next chapter, these are abundantly justified by large throughput gains.

Algorithm1 describes into more detail the methodology. First, we model the Static Control Part (SCoP) in PM. Then, given a splitting factor, we tile the target codes. This is done through the invocation of the Chunky Loop ANalyzer (CLAN) [33] toolchain. Then, we regenerate each tiled block, and manually add the required glue logic. Finally, each derived component is passed to an HLS tool, and the components are assembled

Note how all of the optimizations that can be introduced with the help of the PA are completely independent from HLS directives, which are then optimizations orthogonal to our methodology. Additionally, due to the way the separation of logic and data is carried

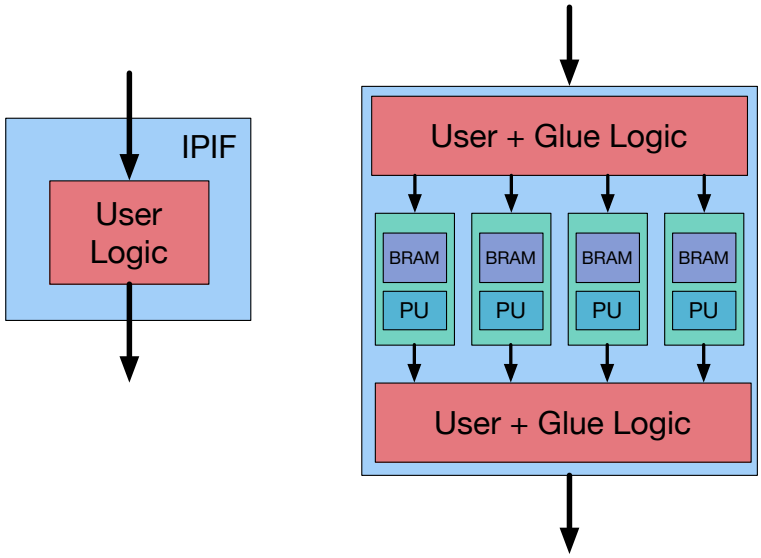


Table 5.1: *Methodology overview. The original accelerator (left) is separated into multiple computational kernels (right), each featuring its local store and related processing unit.*

out, this methodology is orthogonal to other techniques, specifically those related to memory architectures for stencil computation such as [218], the addition of which in our architecture is left as a future work.

As the computational units are self contained components, we can use them regardless from where they are physically located. This enables us to make initial considerations on an FPGA based distributed computing system. Following this technique we are able to separate computation on different processing units and enabling HLS and SLDdirectives further improve the throughput of our solution.

5.3.1 Limitations

As with most state-of-the-art techniques relying on PA, we can only analyze SCoPs. While this limits the generality of the approach, most scientific computational kernels can and are actually described as SCoPs. Moreover, as they generally expose plenty of data-level parallelism, they are also the perfect fit for our approach.

Another limitation in the current process is that while most tools

Algorithm 9: Methodology Pseudocode

```
pmRepr ← CLAN(InputFile)
if pmRepr != regular then
  exit(−1)
end if
deps ← CANDL(pmRepr)
deep ← findParallelDeep(pmRepr, deps)
inputTransf ← tiling with current splitting factor
transformedPM ← CLAY(pmRepr, deps, inputTransf)
cFileNames, tclFileNames, tclArch = []
for all Scop in transformedPM do
  listBlocks ← getBlocksAtDeep(pmRepr, deep)
  for all block in listBlocks do
    block ← glue & synchronizatinlogic
    cFileNames.append(writeCFile(block))
    tclFileNames.append(writeTclFile(block))
  end for
  tclArch = writeArchTcl(listBlocks)
end for
```

are available as open source and allow most operations in the PM domain, the choice of the splitting factor, additional affine transformations and the generation of the glue logic are still manual processes. However, as already argued, affine transformations and splitting factor can effectively and automatically be explored via tools like LetSee [155]. The generation of the glue logic, on the other hand, would require a specific tool for the manipulation of the polyhedral domain and the polyhedral access relations. However, this process can be automated for SCoPs, as order of operations are known in advance.

5.4 Experimental Results

We tested our methodology on multiple kernels derived from the Polybench benchmark suite [152]. Before running them on FPGA we rewrote the kernels in a more HLS-friendly way (like: adoption of adequate bit-widths for involved types, addition of streaming interfaces, and eventually introduction of HLS directives). We tested the methodology on the following kernels:

jacobi2d Jacobi 2D stencil computation. The simplest of the kernels. It exposes plenty of data level parallelism in the central loop, as the core dependencies arise from inter-loop dependencies,

2mm two matrix multiplication kernel. This kernel has an highly data-parallel internal loop. However, it is difficult to further extract parallelism due to Read-After-Write (RAW) dependencies.

3mm three matrix multiplication kernel. A more complex, but analogous to 2mm, kernel.

2dconv 2-D convolution kernel. This kernel features a nested loop with high data level parallelism. This kernel is particularly interesting, as FPGAs become the platform of choice for implementing general filters and convolutional networks (like in [61])

bicg Bi Conjugate Gradient computation, another popular, data-parallel affine kernel.

We tested this methodology by synthesizing and running the resulting designs on a ZedBoard equipped with a Zynq7000 device. While not exactly a high-end device, we anyway observed large relative speedups, that demonstrate the validity of our approach.

We employed the Vivado 2014.3 toolchain and Vivado HLS to generate HDL components.

5.4.1 Experimental Data

We implemented each previously described kernel in three different ways:

Simple HLS The designs derived using this strategy are our baseline. We took the kernel code and added the communication interfaces needed to realize a channel between the main memory and FPGA.

Split-Down Designs obtained this way have parallelized by means of tile only, without relying on any hardware optimization.

Computational Units These designs were generated using our methodology. Base splitting factor is 4. Amount of localized memory depends on the kernel.

Theoretical best This is a theoretical extrapolation obtained by assuming all resources on one ZedBoard, increasing the number of parallel cores as much as possible.

Kernel	Dims	BRAM (%)	DSP (%)	Watt	Time (ms)
Jacobi 2-D	300x300	93	4	1,558	117
2mm	200x200	93	2	1,547	800
3mm	140x140	92	4	1,572	1120
2-D Convolution	300x300, 9x9 conv.	97	4	1,568	45
BiCG	300x300	94	5	1,554	11,75

Table 5.1: *Simple HLS results. Note the heavy pressure on BRAM usage, with poor usage of other resources.*

Kernel	Dims	BRAM (%)	DSP (%)	Watt	Time (ms)
Jacobi 2-D	300x300	97	9	1,66	52
2mm	200x200	N.A.	N.A.	N.A.	N.A.
3mm	140x140	N.A.	N.A.	N.A.	N.A.
2-D Convolution	300x300, 9x9 conv.	97	15	1,707	16
BiCG	300x300	100	18	1,691	3,64

Table 5.2: *Split code. While tiling generally improves the overall execution time, this usually comes at the expense of resource usage: indeed, 2mm and 3mm could not be synthesized using this approach.*

Kernel	Dims	BRAM (%)	DSP (%)	Watt	Time (ms)
Jacobi 2-D	300x300	54	78	1,738	6
2mm	200x200	72	72	1,760	21
3mm	140x140	95	74	1,746	42
2-D Convolution	300x300, 9x9 conv.	45	90	1,790	1,6
BiCG	300x300	96	72	1,856	0,725

Table 5.4: *Theoretical best. Maximal number of splits. This data shows how the synthesized Computational Units approach would perform assuming maximal utilization and bandwidth is not a bottleneck.*

We dimensioned the problem so as to fill, in the baseline ver-

Kernel	Dims	BRAM (%)	DSP (%)	Watt	Time (ms)
Jacobi 2-D	300x300	9	13	1,618	30
2mm	200x200	9	9	1,6	152
3mm	140x140	19	15	1,640	209
2-D Convolution	300x300, 9x9 conv.	9	18	1,682	7,8
BiCG	300x300	24	18	1,742	1,7

Table 5.3: *Computational Units approach. 4 splits. Note how parallelizing (due to tiled components) and localizing data keeps resource usage under control.*

sion, our Zynq7000; the most constrained resource are BRAMs, as the original code allocates all the data as local store. While they are almost completely used, all the other resources – like LUTs or DSPs – are almost unused. Note how this is an inefficient employment of resources, as most logic is wasted (see Table 5.1); however perfectly working, the naïve approach can only be considered the baseline design.

We then created the parallel tiled version, split factor 4. As we need to move back and forth data to and from the hardware accelerator, we opt to synthesize a DMA controller per core in order to send and receive data in a completely asynchronous way. While other implementations are feasible, we chose this solution as it delivers the best throughput, as each DMA controller can transfer data independently from each other. As expected, the generation of parallelized hardware produces better results than plain conversion of sequential code; however, this is not possible in some cases, as data dependencies induce the HLS tools to instantiate a too much local memory. However, execution runtime is better than the baseline, as we see good speedups in synthesizable designs.

Then we designed one accelerator per computational kernel, using the methodology presented in this work. We obtain a better balanced overall design with respect to all other approaches. The source of this drastic improvement is the addition of both the glue logic and the computational units, mechanism that allows us to better exploit the local stores via data (and BRAM) reuse. Obviously, large speedups are obtained against the baseline approach; however, even against the tile-only parallel version we obtain dramatically better results.

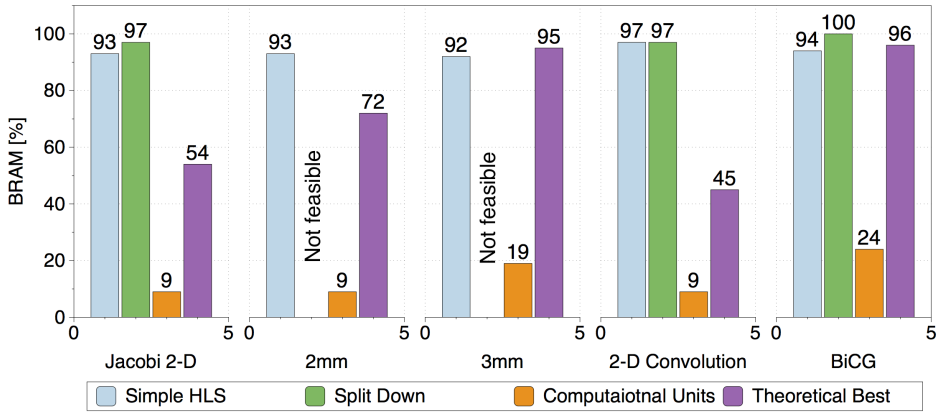


Table 5.2: *BRAM Usage [%]. Note how BRAM usage is scarce in the third approach, even though performances are dramatically better than the other approaches.*

5.5 Conclusions and Future Work

We have shown in this work a novel technique that explicitly isolates data and computation meant to pave the way to further SLD methodologies meant to distribute "units of computation" among an arbitrarily large number of Field Programmable Gate Arrays (FPGAs). We achieve large, near-linear speedups, demonstrating how PM can be employed to distribute a data-parallel workload in a future multi-FPGA based system.

As already stated in Section 6.2, this work is orthogonal to other HLS techniques: given our results and the conducted experience, we believe that future work would benefit from the introduction of a memory architecture similar to that of [206]. Additionally, further work is to be conducted on the choice of the specific affine transformations that might improve the overall process, as well as on the architectures for effective multi-FPGA designs.

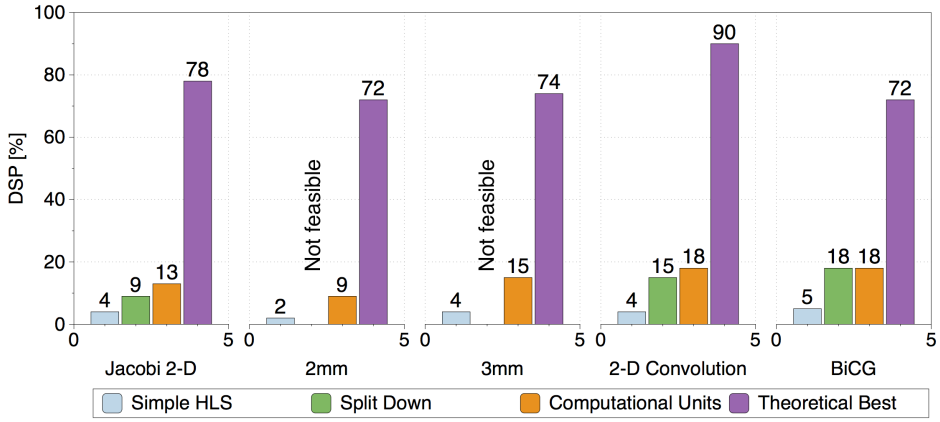


Table 5.3: *DSP Usage [%]. Note how DSP usage is more efficient in the third approach, as we can better paralyze the computational hard part of the kernels.*

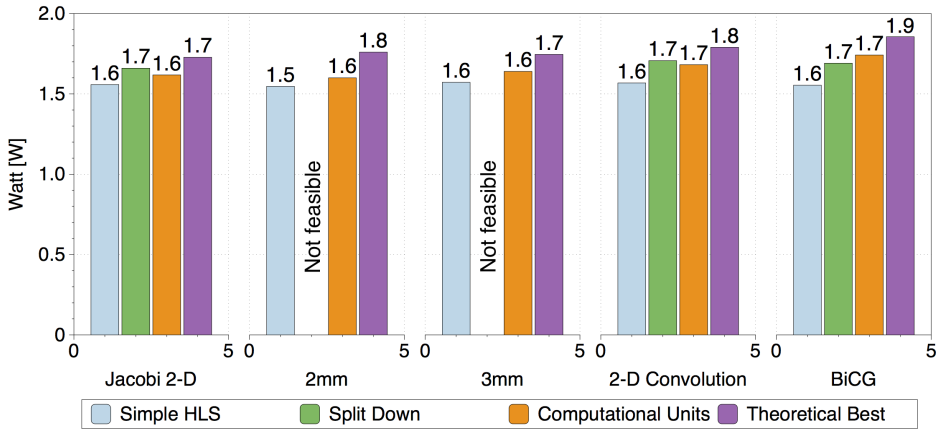


Table 5.4: *Watt Usage [W]. Note how our approach draws slightly more relative power; however, as a much larger throughput is obtained, overall power efficiency is improved.*

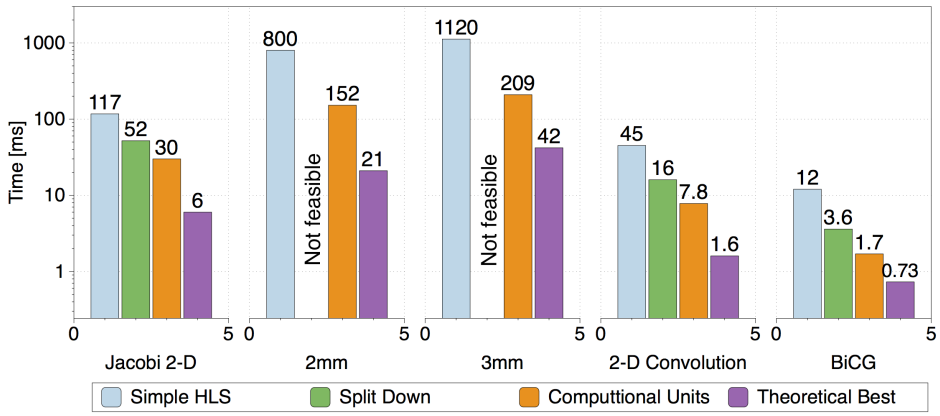


Table 5.5: *Execution Time [s]. Note how execution time is drastically reduced in the third approach. Black columns correspond to designs that could not be synthesized due to excessive resource usage.*

CHAPTER 6

Towards the Optimal Iterative Stencil Loops Implementation

In this Chapter we introduce a novel technique to operate an imperative-to-streaming translation of SANLP codes. This technique realizes a novel, highly effective HLS code manipulation technique featuring the generation of *dataflow-by-construction* hardware accelerators and *full on-chip buffering* of reusable data.

6.1 Introduction

Numerical methods for partial differential equations solving employed in weather and ocean modeling [43, 129], fluid dynamics [73, 179], quantum dynamics simulations [140], heat diffusion [93], geometric modeling [115] and non-equilibrium statistical mechanics [51], but also seismic simulations [170] and cellular automata [141], as well as multimedia/image-processing applications such as [102], gaussian smoothing [184] and Sobel edge detection [23], represent only a fraction of a wide class of applications sharing the same com-

ALGORITHM 1: Generic ISL Algorithm

```
foreach t ⩽ TimeSteps do
    foreach points p in matrix P do
        p ← f_transition(stencil(p))
    end
end
```

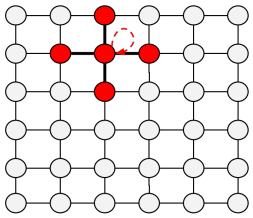


Table 6.1: *On the left, a generic ISL pseudocode. On the right, an illustration of a generic 5-point 2-Dimensional ISL.*

putational nature, in which a series of sweeps are performed over a regular grid, whose points are updated using a fixed nearest neighbor pattern. Algorithms and codes structured this way are called *Iterative Stencil Loops (ISLs)*.

These algorithms update values associated with points on a multi-dimensional grid, using weighted contributions from a subset of its neighbors in both time and space (see Figure 6.1). The fixed pattern of neighbors is called *stencil*, and the function that uses those elements to update an array cell is called *transition function*. An ISL can be generically represented by the pseudocode of algorithm reported in Figure 1 on the left. As already mentioned, the number of algorithms whose computational kernels fall into this category is large and relevant to both industry and science, which is why efficiently implementing them is of great concern.

As they are characterized by a regular computation structure, they are ideal candidates for automatic compile-time analysis and transformation aimed at improving their run time performance. While many state of the art works precisely explore this opportunity (particularly in the multicore/multi processor domains), ISLs *memory boundedness* and *spatial dependencies* limit the scope of current transformation, analysis, or optimization techniques [55, 125, 134].

With *ISLs' memory boundedness* we refer to ISLs' requirement for potentially large transfers of data blocks in order to compute the next state of each element of the problem grid. Additionally, depending on the size of the grid and the stencil, and the degree of parallelization, also processor-to-processor bandwidth might be an issue, as *halos* (i.e.: portions of data processed by a processing unit and required by another to progress the computation) must be transferred, too.

ISLs' spatial dependencies, on the other hand, is the presence of true

data dependencies in ISLs' codes between updated points of the grid between different time frames. Depending on whether further true data dependencies are present inside each time frame or not, automatic parallelization and further aggressive optimizations might or might not be made, making the overall automatic optimization process heuristic and more complex.

Some relevant consequence induced by these properties are:

- The performance of parallelization-oriented techniques is bounded by the available off-chip and on-chip bandwidth, due to the un/loading of the working sets into caches, the synchronization of the parallel units, and halos exchange. Hence, in most cases the achieved performance is far below than the predicted one;
- Techniques designed to take advantage of data locality are not really effective when designing custom accelerators, mainly due to the nature of the computing architectures on which they are applied; a well designed pipelined architecture, given the regular structure of ISLs codes, is comparable in structure and performance to the tiled one;
- Although the employment of custom logic explicitly designed to accelerate a specific ISL is promising, it is in general a hard task.

6.1.1 Contributions

To the best of our knowledge, no work addresses all the presented issues *at once*, as it either focuses on a subset of them or fails to properly consider bandwidth and memory as *the* limiting factor of a system designed to scale. Aim of our work is to *simultaneously* address all these issues at once with the definition of a proper ISL-centric methodology.

Therefore, our contribution is:

1. A streaming architecture implementing a single stencil time-step able to realize full data reuse with the minimum on-chip memory requirements while allowing for multiple concurrent accesses, the Streaming Stencil Time-step (SST); we realize it as a distributed architecture exploiting the inherent parallelism of a FPGA;

2. A scalability-oriented technique able to deliver quasi-linear speed-up thanks to constant bandwidth requirement, namely SSTs *queuing*, our source of parallelism;

3. An architecture and a methodology to exploit it – a *design automation flow* – to automatically implement ISLs with the proposed hardware accelerator.

The remainder of this Chapter is organized as follows. Section 2 thoroughly reviews related works and their limitations. Section 3 precisely delineates the contribution of this work to the state-of-the-art. Section 4 is an introduction to the overall methodology. The SST, the core computational element of the proposed system, is thoroughly described in Section 5. Details on how the system can be effectively scaled using the *SST queuing* technique is presented in Section 6. Section 7 provides experimental results, both of the single SST component and the queued ones, with comparisons against CPUs. Finally, Section 8 concludes the thesis and elaborates on future works.

6.2 State-of-the-Art

ISLs have been extensively studied in state-of-art-works. We divide the section into three parts to introduce the most relevant techniques and compare them against our solution.

6.2.1 Tiling Based Optimizations

Tiling, also known as *blocking*, is a code optimization technique employed to both enhance data locality and exposing parallelism. This technique has been exploited in a number of different ways, and performed in both spatial – when possible – and temporal dimension [163], as we show below.

Single iteration tiling. The earliest and simplest tiling-based technique usually employed both in CPU and GPGPU code optimization consists of applying conventional loop blocking to improve cache reuse. In this case, a single time frame (*i.e.* a single iteration) is partitioned into smaller blocks, allowing points that are close in space to remain in the cache when used, thus allowing to update them together, improving locality [110]. This technique has also been exploited to distribute the computation to multiple processing elements,

in order to parallelize the computation of next-state points within a single iteration [92], also leveraging specific APIs such as OpenMP. However, tiling across multiple processsing elements potentially incurs in heavy off-chip and on-chip bandwidth requirements, as stencils along a tile's boundary require values that were previously computed by other processsing elements, increasing communication and synchronization between them. An effective technique to overcome this issue is the one known as *ghost zone optimization* or *overlapped tiling* [117, 134], which consists in the enlargement of the tiles with ghost zones, *i.e.* the overlapping regions between tiles, replicating some computations but nevertheless reducing communication and synchronization. Although this technique might mitigate communication issues, an improper selection of the ghost zone size may result in even worse performance with respect to no optimization at all [107]. Additionally, generic processors deliver poor power efficiency figures compared to custom logic accelerators, as most of their area is dedicated to coping with irregular code computation. As a final remark, dealing with ISLs with spacial dependencies between grid points forbids the application of this optimization scheme for parallelization purposes, and performance are usually degraded even with cache optimizations as outlined in [207].

Time skewing. In this tiling scheme, multiple iterations are *collectively* partitioned into blocks: with respect to single-iteration tiling, multiple iterations are computed as part of each tile. The reason beyond the application of such a strategy (specifically in CPUs and GPGPUs) is to exploit temporal locality, too, thus increasing the overall data reuse factor. However, in order to make tiling legal, loop skewing [210] along the time dimension is required. In fact, as point update occur in both spatial and temporal dimensions in each block, they must shift their collection of points backward on the time dimension to respect temporal dependencies induced by the ISL, *i.e.* transform dependency distances into non-negative values [168], resulting in a loss of inter-tile concurrency (skewing introduces inter-tile dependencies in the spacial direction). While it might seem that this scheme always delivers better performances than the simple single-iteration version, it really depends on a careful selection of the skewing factor [163], as well as on the form of the tile [168,188], which can be a major concern especially on FPGAs [219]. Whit respect to the previously mentioned strategy, time skewing can provide better cache hit rates and effectively reduce pro-

cessor idle time caused by the ISLs' memory boundedness [212] in CPUs and GPGPUs.

As for the previous tiling strategy, even in this case blocks distribution among different processsing elements is possible [21], but likewise single-iteration tiling, it requires explicit synchronization between them, since a block must wait for its neighbors to complete in order to have enough data to start. As a consequence, rather than a purely parallel execution, in timeskewing blocks are executed in a pipeline fashion.

A possible solution to mitigate time skewing's adverse effects when tiling along multiple iterations is proposed in [137], where *code transformation* is performed to fuse the stencil loops together in order to reduce the number of reads and writes, and increase both the computational intensity and data reuse via loca buffers. While this is an effective approach to cope with ISLs' memory boundedness, scaling their approach is not straightforward, as they focus only on data reuse efficiency rather than scaling out the system. A very similar technique has been developed in [54], in which a domain-specific compiler is proposed, namely *Caracal*, able to perform unrolling of the time loop and fuse accordingly the stencils, with comparable effects as of [137].

Wavefront parallelization. Instead of pipelining the execution of time-skewed blocks, wavefront parallelization dictates that blocks must be scheduled collectively in a *wavefront* fashion [180, 196, 208]. Blocks are arranged in a way that on the time dimension the computation blocks are independent from each other, thus not requiring synchronization. Although in [163] this scheme has been explicitly defined as the one in which *multiple* blocks are *scheduled together*, this class can be easily extended to the case in which only *one* block implements a single iteration. Indeed, this is exactly the behavior exhibited when tiling is *only* applied on the time dimension [146, 175, 176]; this approach is promising as it has been proven to scale [146, 175, 176] with low communication overhead. We draw inspiration from this work and largely extend it to be able to build a dataflow architecture customized around the specific ISL workload.

6.2.2 DSLs Based Optimizations

The exploitation of Domain Specific Languages (DSLs) and ad-hoc frameworks has been explored as an interesting way to drive

compile time optimizations by exposing specific, custom problem semantics to compilers. While General Purpose Languages (GPLs) are the dominating software development tools in HPC, the lack of specialized language constructs and semantics to serve domains such as ISLs is a limitation, since most of the times they do not allow to express a problem in a manner that allows compilers to explicitly and directly manipulate the code.

DSLs are certainly interesting, as they allow designers to define a problem in a way that some features are explicitly signaled to the compiler, allowing for whole kinds of aggressive manipulations, optimizations, and specific code generation techniques. However, this comes at the cost of losing broad applicability, as they usually allow the expression of a fairly limited set of algorithms.

Among ISLs-oriensted DSLs, PATUS [55] is able to achieve a high performance by means of *auto-tuning*, targeting a different hardware architecture, while Pochoir [193] provides a C++ template library based on a divide-and-conquer skeleton which is then translated into Cilk [44], a C/C++ extension designed for multithreaded parallel computing. ExaStencils [122] employs a direct mathematical formulation (ExaSlang) of the problem, and through a series of steps of transformations, included a wide range of polyhedral model-based optimizations, generates target code in a specific language, which by now is C/C++, but in the future could be extended to other languages. DeLite [190] abstract from Scala with the aim of making stencil programming easier, and use meta-programming to construct an Intermediate Representation (IR) of the problem and compile to a large number of languages, so that it can easily target heterogeneous hardware. In [217], a single mathematical formula is used to implement 3-D stencil codes on GPGPUs, via auto-tuning and automatic target code generation, and GPGPUs is also the target device of [107], in which low-level code is generated, starting from an abstract representation, by trading an increase in the computational workload for a decrease in the required global memory bandwidth. In [209] a single high-order function specified in Haskell, and specifically in CλaSH [26], a functional HDL able to translate plain Haskell (with some restrictions) into synthesizable VHDL, is used in combination with a series of transformations to generate hardware accelerators.

As a final consideration, although using DSLs can lead to good performance, in HPC and industrial-grade high performance em-

bedded systems this is not common at all, as GPLs are preferred due to their versatility, ease of use, and vast availability of optimized libraries and components. Additionally, designers are usually familiar with imperative languages like C or C++, which is a major limitation towards the adoption of DSLs, which employ custom (and non necessarily imperative) semantics; indeed, most available HLS tools ship with support for C and C++. For these reasons, we adopt a subset of C as our input language, and specifically a fragment of it analyzable by means of the polyhedral model, in order to both support an industrial-grade language, and to be able to heavily analyze, restructure, and manipulate input codes in order to optimize the resulting implementations, as we thoroughly describe in Section 7.4.

6.2.3 Custom architectures

When designing custom hardware, FPGAs can offer both high flexibility and sustained performance with high energy efficiency, often orders of magnitude better than other hardware platforms, depending on the target workload. In fact, an increasing number of works are focusing on exploiting FPGAs to implement ISLs with the development of custom hardware, finely designed to efficiently leverage the regular structure of this class of algorithms. Specifically, an adequate analysis and design of custom hardware has been proven useful in mitigating the memory boundedness issue of stencil computations.

In [169] for instance, a generic tunable VHDL template has been proposed to parallelize 3-D stencil computations. Their work uses the so called Full Buffering [126] in favor of Partial Buffering (which is a strategy where solely the data needed by the current computation is stored to minimize memory consumption) a technique in which data is stored on the on-chip memory until all the computations depending on it have completed, showing that the increasing number of available resources in modern FPGAs allows to obtain very good performances. However, this work has not been thought of as streaming in nature, and most importantly it was not thought of as a scalable solution (i.e.: runnable on multiple processing elements with adequate memory and bandwidth considerations, which we do in our work).

In [59] the polyhedral model is employed to take advantage of

the stencil access pattern and perform non-uniform memory partitioning in order to generate a custom microarchitecture, streaming oriented, which is proven to be optimal with respect to memory usage, since it allows Full Buffering with the minimum number of reuse buffer banks and minimum buffer size. This architecture has only been simulated, without actual bandwidth and memory consumption figures, the main bottleneck when scaling an architecture to multiple computational elements. Additionally, the case in which the computation has more than one input channel (a common situation in ISL computation) is not covered at all. There are however two required considerations with respect to [59], that the proposed work instead properly addresses (1) the proposed architecture is not able to deal with ISLs with spatial dependencies among points updates, (2) they don't validate the proposed architecture in real test benches. Consequently, they do not provide any insight on how well the solution performs, also considering that estimated results do not effectively take into account constraints such as available bandwidth.

The work of [177] consists of replicating the architecture demanded to perform one time-step a number of times putting them in cascade, *i.e.* the output of one architecture is the input of the next. The hardware accelerator is a composition of soft-processors that must be explicitly programmed, hence it is a totally different approach with respect to the one proposed in this work where an SST is an automatically derived architecture. This work is the first proposing a methodology to construct the queue automatically, along with some concepts, such as the *queue looping*, which are completely novel. In [116] 2-D stencils are addressed using *ScalableCore*, a system composed of multiple, low end FPGAs, connected in a 2-D-mesh. To efficiently exploit such an architecture, the stencil computation is tiled and each computational block is assigned to an FPGA.Their work proves how an FPGA custom architecture delivers an improved power efficiency than traditional computing devices. However, the isolation of computation on multiple computing elements effectively reduces data reuse by a factor proportional to the number of elements, which is a disappointing property when the goal is to scale out the system. Moreover, *ghost zones* must be properly synchronized among processing elements, leading to unnecessary (as we show in Section 7.4) increase in on-chip bandwidth consumption.

In [182] a memory architecture is developed to implement symmetric 3-D stencils, *i.e.* of the form of $n \times (n+1) \times n$, which features

FIFO queues for input and output streams (one for each dimension), a *data engine* (the *front-end*) which prefetches data, a *compute engine* (the *back-end*), which consists of multiple instances of the computation unit, and a *control engine* responsible for synchronizing the flow of data in the whole architecture. While this work is interesting for the customized approach to 3D stencil computation, the applicability is limited to this class only; additionally, no explicit scaling mechanism is reported. In our work, we extend the applicability of our methodology to N-dimensional stencils and explicitly define a scaling methodology.

6.3 This Work's Contributions

From the context described in the previous section it is evident that:

- The obtainable performance of parallelization-oriented techniques can be bounded by the available bandwidth, due to the increase in bandwidth demand consistently to the increase in parallelism, but also for the consequent need for synchronization of the parallel units. Hence, in most cases the achieved performance can be far below the theoretical peak.
- The techniques designed to take advantage of the data locality are not effective, mainly due to the inadequacy of the computing architectures on which they are applied.
- The scalability of ISLs in large-scale clusters is hard to achieve and nevertheless the performance does not increase linearly with the scaling.
- Although the employment of custom logic explicitly designed to target ISLs could be promising, it is in general a hard task, and it definitely needs automation in order to ease the process and make it accessible to a broad user base.

To the best of our knowledge, the existing works do not address all the presented issues, as they instead focus only on subsets of them, resulting in suboptimal solutions that are not able to efficiently cope with all the challenges posed by the ISL implementation. The aim of this work is instead to address all the presented issues at once

with the proposal of an hardware accelerator specifically designed to target ISLs, indeed:

- We realized a distributed microarchitecture which exploits the inherent parallelism of the distributed nature of an FPGA. Our source of parallelism comes also from the employment of a technique which enables a pipelined execution of multiple time-steps within the accelerator, allowing to perform concurrently multiple time-steps in one pass. The robustness of this technique comes from the fact that the increase in performance is achieved without an increase in bandwidth demand, therefore it is always possible to increase the throughput, even in the case where the available bandwidth is very limited.
- The proposed memory system is designed to allow multiple concurrent accesses – that is exactly what is needed in ISLs as they compute using a nearest neighbour pattern, and avoid resource contention, a practical issue in the case of FPGA. Another important peculiarity of the memory system is the fact that it is able to deliver full data reuse, thus reducing to the *minimum* the amount of required communication with the off-chip memory, realized with the *minimum* achievable on-chip memory requirements.
- The previously cited technique that enables the execution of multiple time-steps in one pass ensures linear scalability, with constant bandwidth requirements. This allows to easily scale without incurring in performance degradation, and can also enable scaling over multiple FPGAs nodes, solving effectively the problem of scaling in large clusters.
- The proposed hardware accelerator can be directly derived from an imperative specification of the ISLs, *e.g.* an algorithm written in C/C++. We indeed proposed a *design automation flow*, which employ the PM to achieve this goal.

A detailed overview of the proposed solution will be supplied in the next section. Let us however briefly summarize the thesis contributions. In practice, we provide:

1. A streaming-based microarchitecture that implements a single stencil time-step able to realize full data reuse with the minimum on-chip memory requirements, the SST;

2. A scalability-oriented technique able to deliver pseudo-linear speed-up, namely SSTs *queuing*;
3. A methodology – a *design automation flow* - to automatically implement ISLs with the proposed hardware accelerator.

6.4 A Scalable Streaming-based Microarchitecture for the Automatic Implementation of ISLs

The work proposed in this thesis targets ISLs implementation employing a custom hardware accelerator. For this purpose, we developed a streaming microarchitecture aimed at performing a single ISL time-step, which we called SST. The entire accelerator is represented by the composition of multiple SSTs in a *queue fashion*. We then also proposed a *design automation flow*, to automate the SST derivation and the queuing process.

In the rest of this section we provide the key points, a description of the accelerator in all of its components, a set of constraints for the input code of the proposed solution, and a comparison with existing works.

6.4.1 Fundamental Principles

Let us first expose the fundamental principles on which the proposed work rests its foundation.

Streaming Computation

Within the context of regular computations, such as ISLs, a streaming paradigm is undoubtedly a well suited choice. The ability to perform complete compile-time analysis allows to determine precisely the data flow, and as a consequence, to arrange the computation in the most effective way. A streaming computation model is indeed a *data-centric* model, where the focus is on constant data flow, granting high throughput, but nevertheless keeping low the amount of needed resources, especially when the underlying architecture enables this kind of optimization. In the case of ISLs, where the update of each grid point requires a number of concurrent reads, this approach can avoid – or at least limit – the problem of memory boundedness effectively reducing resource contention. Also, the distributed nature of a streaming model fits perfectly the distributed

nature of a configurable architecture such as FPGAs, enabling the exploitation of the inherent parallelism of those devices.

The reader may refer to section 4.2 for the technical details about streaming systems on FPGAs.

Scalability

In assessing the quality of a system, scalability is absolutely an essential parameter. The scalability issue is actually a hot topic, as indeed a large amount of work, theoretically valid, are actually suffering from a limited scalability or in the worst cases they do not scale at all. Our work is focused on scalability, which is in fact addressed in two different and complementary ways:

- The accelerator itself is based on a scalable architecture. SSTs are connected in a linear array that constructs a deep pipeline. Because the depth of a pipeline does not influence the bandwidth requirements, we can increase the computing performance with a constant memory bandwidth by connecting more SSTs for a longer queue, increasing consequently the throughput.

- For large problem sizes, whenever the available on-chip memory resources are not enough, the communication channels can be removed and substituted with an off-chip memory interface, thus increasing the bandwidth consumption while reducing the on-chip buffering requirements.

Optimal Full Buffering

When the memory resources were so limited that memory systems on FPGAs allowed storage of only a very small amount of data, Partial Buffering (PB) was the only way to go. The principle beyond PB is that data is fetched from external memory only when it is needed, which means that, if needed multiple times, the same data is transferred more than once. This technique allows to keep low the resource usage, but also the overall performance, as it implies repeated reads for the same data from off-chip memory, consequently resulting in the waste of clock cycles.

Modern FPGAs have now enough resources to allow, when the computation is performed on reasonable problem sizes, the employment of Full Buffering (FB), a technique in which data is read only

once and stored on the on-chip memory until all the computation relying on it has completed. The advantage of a FB scheme is that, at a cost of an increase in scratch-pad memory requirements, the off-chip traffic is reduced to the minimum.

An SST is able to perform FB in an *optimal* way, employing the PM to perform non-uniform memory partitioning of the input stream. The compile-time analysis allows to compute the minimum size of the reuse buffer for a data array, which is indeed equal to the maximum lifetime of any element in the array itself. In this way an SST can deliver FB with the *minimum* number of buffer banks (represented in the architecture as communication channels), were each of them have also the *minimum* possible size.

Wide Applicability

A lot of existing works focus only on ISLs *without spatial dependencies* between grid points within the same time-step (see section 4.6.1), mainly because it is difficult to extract parallelism from those algorithms. Hence, they intentionally limit their applicability, as their solutions are not suitable for this kind of ISLs.

We instead treat indiscriminately both ISL types, as our methodology leverages a streaming-based computation and the performance gain is given by the pipelining of multiple SSTs. Our source of parallelism is indeed implicitly given by the distributed organization of the mircoarchitecture, which in turn takes advantage of the distributed nature of FPGAs.

As last remark, even though a proper restructuring of the input stream could remove this limitation, it must be said that our methodology does not target ISLs with *periodic boundary conditions*. This is indeed a limitation shared by a lot of available works, since those kind of ISLs are not as common as the one with constant boundaries.

Automatic Process

As stated in the previous section, hardware design is an hard task and, if it is done *by hand*, also error prone. For this reason we proposed an *automated design flow*, able to generate the accelerator directly from the input source code. This flow will be detailed in the next chapter.

6.5 A General Overview of the Proposed Microarchitecture

Hardware acceleration is one of the techniques used to improve performance of a computing system. It consists of offloading the general purpose processor from the computationally intensive part of a given algorithm, that can rely on computer hardware specifically designed to perform those computations. The proposed microarchitecture precisely embodies this logic, as it is indeed an hardware accelerator. This means that it requires a *host processor* to drive the execution and control the in and out data flow.

From a general perspective, the proposed microarchitecture can be viewed, at every level of granularity, as a composition of *independent* modules, that communicate over FIFO channels and employ *blocking reads* and *writes* to manage the data flow and ensure its correctness.

At the top level, the accelerator consists of a series of blocks arranged in a queue fashion, each of which is responsible for the execution of a single ISL time-step. Those blocks are called Streaming Stencil Time-step (SST), a name we chose as it recalls exactly their functionality. Since the microarchitecture is streaming-based, data flows from an SST to another as soon as it is produced, resulting in a pipelined execution of the entire computation. In some occasions the SSTs data flow can be managed by an additional module, which is always aware of the progress of the computation as well as the total number of iterations to be performed, which we called *mux*. This happens in two cases (which can also occur together):

- the number of SSTs - and hence the corresponding number of time-steps - of the hardware accelerator are *not* an exact divider of the total number of iterations. In this case the *mux* is responsible to break the computational flow when the total number of time-steps of the ISLs is performed, and redirect the output to the off-chip memory.
- the queue length is large enough to be able to cycle the data flow, redirecting the output stream of the queue back to the queue itself, instead of transferring it back to the off-chip memory. We called this condition *queue looping*, which will be further detailed in the next section. In this case the *mux* is responsible to break this loop when the total number of time-steps has been executed.

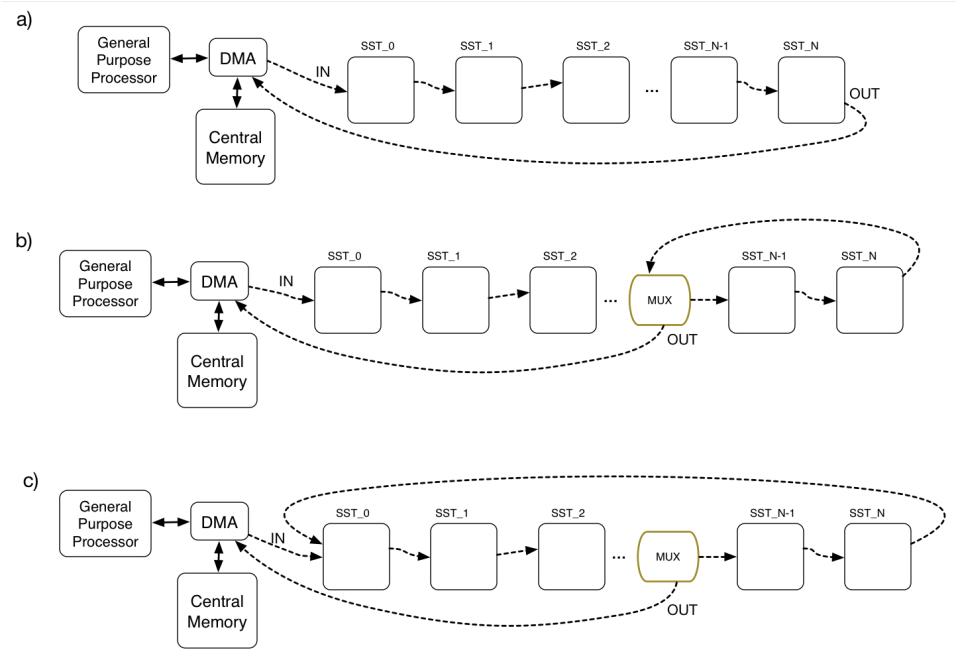


Table 6.2: *The high level scheme of the proposed hardware accelerator. The three different versions represent the three distinct described cases: the first (a) is the standard case, the second (b) is the case in which the queue length is not an exact divisor of the total number of ISL time-steps, the third (c) is the case in which there are enough available resources to enable queue looping.*

Now that we have described the accelerator from a high level, the only thing that remains to detail is how an SST is actually implemented. We already claimed that an SST is demanded to execute a single ISL time-step, let us now describe its internal components.

As first thing, an SST *in general* has one input stream and one output stream. In the case in which the ISL updates grid points employing constants or other arrays, the input streams are obviously more than one. The components within an SST can be divided into two main categories, the first being the *memory system*, the second being the *computation system*.

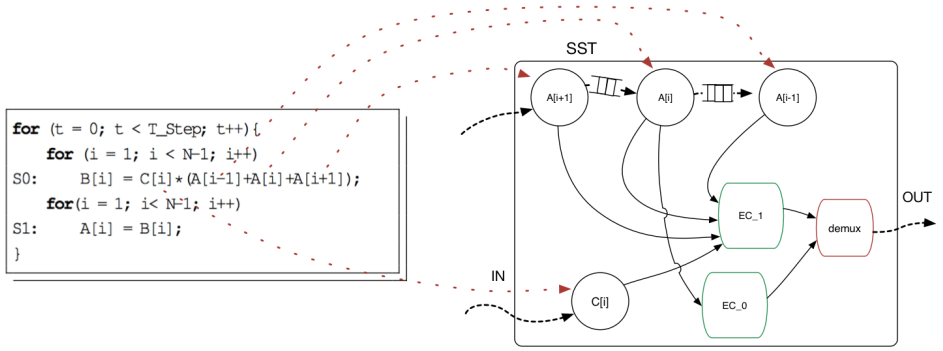


Table 6.3: *A general scheme of an SST.*

6.5.1 Memory System

This part of the SST consists of a series (or one, when the input is just the single stencil array, *i.e.* the one updated from the ISL) of *chains* of modules connected by FIFO channels, one chain for every distinct input array, all responsible to feed the computation system with the needed data. Each chain receives a single data stream, which is indeed the array itself, and the modules within the chain represent the different read array references. Trivially, if the array reference is unique, the chain is made up of a single element. These modules are indeed the one actually responsible of sending the data, as in fact they read any existing data element from their preceding FIFO and send the data element to the successive FIFO as well as to the computational system. From a high level perspective, this arrangement can still be viewed as a single stream, from were each module filters data only when needed, which is why we called them

filters. The chain-like organization of filters ensures that the data is read only once and at the same time allows more concurrent accesses, realizing also the optimal FB.

6.5.2 Computation System

The computation system is composed of a series of modules that perform the actual computation taking data from the memory system. However, there are some further considerations to make to better understand how they are arranged.

First of all, given the fact that ISLs update grid points using a nearest neighbour pattern, it is evident that there is always the presence of the boundary to take into account. This condition can cause performance loss when hardware accelerators are employed, as the host processor could be forced to waste time to reconstruct the array from the output. For this reason, our SST consider the boundary in an explicit way. This is indeed also a prerogative for the SSTs queuing. In fact, since the accelerator consists of a chain of replicas of a single SST, it is obvious that within an SST both the output and the input must be of the same form. To accomplish that, the last computing module will always be decomposed into two parts: one demanded to compute the ISL output, which we may refer to as *computation part*, and one which transfers the boundary of the grid from the memory system (without applying any computation, as there is none insisting on that portion of the grid). To ensure that the output stream is rearranged in the exact same form of the input, we inserted an additional module, called *demux*, whose function is precisely the one just stated.

There is also another possibility, that can lead to a further decomposition of a *computation part*, that is the presence of spatial dependencies between grid points. This situation will be addressed in detail in the next chapter, however we anticipate that the computation part will be decomposed in more than one *equivalence class*, that realize the computation considering the presence of these dependencies. A *demux* will be added also in this case, for the same reasons as of the boundary.

We previously claimed that an SST has *in general* a single output stream. This is true in most cases, but not when performing queuing with an ISL that takes multiple array as input, *i.e.* which has variable coefficients. In fact, to provide the data required to all the SSTs, ad-

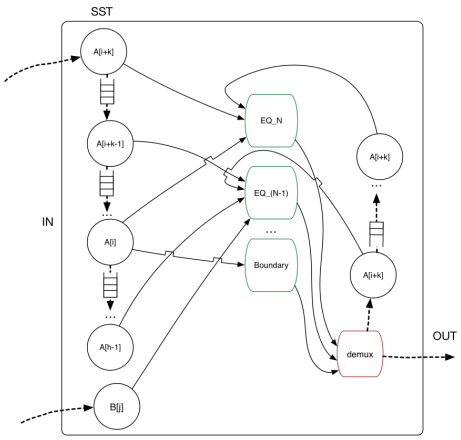


Table 6.4: *An SST for ISLs with spatial dependencies.*

ditional streams must be added, in order to transfer the input data within the queue. In this case then, the *filters chains* refer to arrays which are not the output one are equipped with an additional communication channel, used to drive those data to the next SST in the queue. This is obviously not true for the last SST.

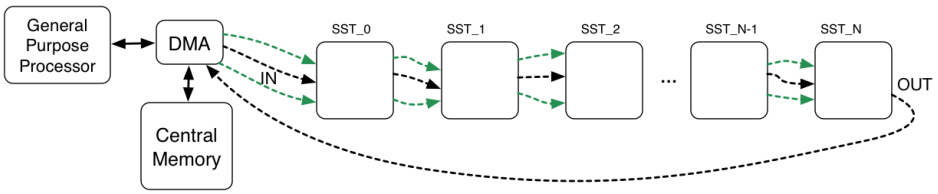


Table 6.5: *An example of the accelerator for an ISL with multiple inputs. The green arrows represent the additional streams. As described in the text, the last SST has only the actual output stream.*

6.6 Some Considerations on the Input Code

We already stated in section 6.4.1 that our solution targets indistinctly ISLs with or without spatial dependencies. There are however some considerations to make about the input code to allow the proposed *design automation flow* to work properly:

1. The algorithm must be specified in an imperative form, *e.g.* C/C++.

Also, it must fall into the category of SANLPs, which is indeed the case for nearly every ISL;

2. There is virtually no limit on the input problem size. Even if the available resources are not enough to handle large arrays, there is always a bandwidth/buffers trade-off that can be made to solve the issue. This case will be further inspected in the next chapter.

3. Even though multiple inputs are allowed, the actual output must be the only stencil array, *i.e.* the one updated from the ISL. In a nutshell, this means that whenever the ISL contains more than one statement, they must be assigned on the same array (the stencil one though). The only case in which statements with assignment on different arrays can happen, is when those statements are in a dependence relation, *i.e.* the array updates of one statement S_i are read subsequently by another S_j. Even in this case the actual output is only one, indeed the array updated by the statement S_j. Hence, array updates of S_i - that can be thought as "intermediate" results - which are not used by S_j, will be still present in the internal data flow of an SST, but not forwarded.
   Such a restriction is indeed necessary to derive effectively the SST. However, it is important to notice that this condition is not at all restrictive, as loop nests that do not have this *single output* feature are not proper ISLs.

4. Conditionals which are affine functions of the time dimension indices are admitted, but require a pre-processing phase, as will be described in the next section. Instead, conditionals on the array dimensions indices are not allowed, as indeed a code with such a structure would not be a proper ISL.

5. Array sizes are inferred by the polyhedral analysis. This is perfectly possible, given that the stencil array accesses map their data space also on the boundary. If the array is bigger than the computed dimension - loop nest and boundary conditions - the *design automation flow* should be assisted with additional information, *e.g.* specific pragmas. The situation in which arrays are bigger than the computed size is however *very* unlikely to happen, as boundary are present only to ensure the algorithm correctness. In the general case, there is no need to

have boundaries bigger than the one employed in the computation, as they would be unnecessary information, and by the way also a waste of memory space.

6.7 A Comparison with Existing Works

The work proposed in this thesis, from a high level perspective, can be analyzed from three different point of views: the automated PM-based *C-to-FPGA* flow, the streaming-based *SST microarchitecture* that targets ISL with a memory system able to achieve full data reuse, and the exploiting of the time-iterative nature of ISLs with the *SSTs queuing* to overcome the memory boundedness. The goal of this subsection is to compare the proposed work with the works that, as far as we know, appear to be the leading in each of the three different aspects.

6.7.1 PM-based C-toFPGA flow

Although HLS have seen an intense evolution, as already described in section 4.3, such that today's HLS tools are capable of generating high quality Register-Transfer Level (RTL) code for a wide range of input programs, they still lack the ability to exploit all the available performance enhancement opportunities, especially for SANLPs. In particular, the essential limitation is given by the absence of a structured approach to efficiently manage data movements from off-chip to on-chip memories, which by default are completely left in the hand of the software designer. The PM can be effectively exploited to overcome this issue, and in fact a number of C-to-FPGA frameworks have been proposed, in particular [124, 159] and [219], which employ the PM as optimizing engine. The aim of all these works is to mask the off-chip transfer latency managing to *intrinsically* overlap communication and computation. However, they are not always able to achieve the desired results, mainly because of the way they use HLS. As a matter of fact, they use HLS only as a back-end for their optimizations, instead of focusing on the real issue, namely the production of an efficient accelerator which leverages the real capabilities of the underlying hardware.

The work of [159] uses tiling hyperplane transformations to expose data locality as much as possible, and then carefully manages on-chip buffers to enable data reuse and pre-fetching. The gener-

ated code is then further optimized to be used for HLS. This work has been implemented in a toolchain, named *PolyOpt/HLS*. PolyOpt/HLS is able to realize data reuse only among subsequent iterations of a loop, although for a given loop nest the depth to which data reuse is exploited - *i.e.* which two successive iterations are used - can vary. This is indeed a limitation, since the framework does not necessarily capture all the reuse potential in a loop nest. In particular, reuse between two non-consecutive iterations is not exploited at all. Even though they claim that this should not be a huge limitation, this is indeed not true, since when access patterns are of the form of ISLs, their technique can fail completely the task of alleviating the memory boundedness issue. Also, when reuse opportunities are only between non consecutive iterations, the quality of their results can be unsatisfactory with respect to the goal of achieving efficient data reuse. Although in the later work of [124] PolyOpt/HLS has been extended with optimizations tailored to solve the resource contention on memory bank ports and achieve an initiation interval of 1 clock cycle on pipelined kernels, two points of failure of the initial work, they admit that with certain kind of data access patterns they still fail to achieve optimal results.

In [219] the authors extend the state-of-the-art framework *PoCC* [12] (a framework for polyhedral optimizations which wraps all the most relevant state of the art tools and libraries) in order to:

- Use their PM-based methodology to extract inter-block and intra-block parallelism and pipelining,
- Produce HLS ready code with all the needed directives,
- Generate the communication interfaces (generally FIFOs) between computation and communication blocks.

Their methodology consists of a set of loop transformations to obtain desired data dependencies between iterations (unimodular transformation), they then utilize a cost model to estimate which transformation is the best in the application context - FPGA resources, type of dependencies and communication costs - and produce the corresponding scheduling. This work is however restricted to the case in which the loop nest dimensionality and array dimensionality are equal for all sets of blocks in the program. This is actually a restriction that dramatically limits the applicability of the proposed

methodology, as for instance we experimented that *all* the benchmarks of PolyBench/C [152], the benchmarking suite for PM-based optimizations, cannot be treated with the proposed framework. The only case in which it could be applied is for a subset of the ISLs benchmarks, namely *adi*, *jacobi-1D*, *jacobi-2D* and *seidel*, and only if the outermost loop - *i.e.* the time dimension - is removed from the original code. They essentially claim that their methodology is in general applicable to SANLPs, but the reality is that it can only be applied to a very small subset of them, not even all ISLs.

Even though our application domain is smaller - but not that much - with respect to the entire class of SANLPs, an aspect that must be considered for the comparison, there is an essential difference between the two aforementioned C-to-FPGA flows and the one proposed in this thesis. We indeed employ the PM *not* to transform the input source code to be "HLS-friendly", but instead to realize an hardware accelerator able to exploit efficiently the available hardware resources and perform optimal FB. In our case, the HLS is not our target, it is instead a link to connect the polyhedral framework and the hardware design.

6.7.2 SST microarchitecure

There are essentially two architectures that can be compared with the one realized from the SST. The first comparison can be made with the work in [59], which by the way has been a starting point of the one proposed in this thesis. Indeed, from a functional point of view the *chains* within the SST's memory system share some similarities with the working principles of [59], even if technically they are implemented differently. There are however two observations that have to be made, since the work in [59] is lacking in two aspects, that the proposed work instead properly addresses:

- the proposed architecture is not able to deal with ISLs with spatial dependencies among points updates, *e.g.* the Gauss-Seidel method, for which the PolyBench/C version has been employed as benchmark in the experimental section.

- they never validate the proposed microarchitecture in real test cases. Consequently, they do not provide any insight on how well it performs, considering also that estimated results do not

take into account practical constraints such as the available bandwidth.

The second comparison can be made with Maxeler [9]. Maxeler is indeed an FPGA-based heterogeneous system, where the accelerator has to be implemented with a dataflow specification, *i.e.* a Dataflow Engine (DFE). Maxeler's computing system includes CPUs and DFEs, and DFEs configurations are created using Maxeler's MaxCompiler. To create applications exploiting DFE configurations, an application must be explicitly split into three parts:

- Kernel, which implements the computational components of the application in hardware.
- Manager configuration, which connects Kernels to the CPU, engine Random Access Memory (RAM), other Kernels and other Dataflow Engines via a custom interconnection (MaxRing).
- CPU application, which interacts with the dataflow engines to read and write data to the Kernels and engine RAM.

From an architectural point of view, the structure of our accelerator is similar with the one obtainable with Maxeler - a Maxeler's DFE - on the specific application domain of ISLs. There is however an essential difference between Maxeler and the work of this thesis: in the case of Maxeler, the software designer *must* have a deep knowledge of the Maxeler system, of the Maxeler language, which is an extended version of Java, called MaxJ, and nevertheless a deep knowledge of the general structure of a dataflow architecture, as it *must* specify the accelerator behaviour in an explicit way. Hence, there is a learning curve and a required expertise that is but easy to attain: being able to implement complex program can be an hard and time consuming task. On the other hand, our methodology is able to extract the accelerator from plain C/C++, with the restrictions specified in section 6.6, automatically.

6.7.3 SSTs queuing

The exploitation of the time dimension in order to increase the performance is not a new idea, there are indeed a few works in which this is done effectively. The key idea is to exploit the iterative nature of ISLs and the temporal locality in order to reduce the

amount of communication with the memory, resulting in an alleviation of the memory bandwidth issue. There are two techniques which employ this idea in two different ways, that can be thought of as "software" and "hardware".

In the software version the original ISL is rewritten to merge two or more time-steps into a single update by expanding the stencil formula along the time dimension. This is done in both [137], where the target is hardware design, and [54], where the target is canonical CPU-based architectures. In [137], the code restructuring can however lead to ports contention on memory banks, due to the enlargement of the stencil windows and the resultant increase of required concurrent accesses on the memory banks, a problem that cannot occur in the case of an SST where the memory system is exactly designed to allow multiple concurrent memory accesses. In [54] the lack of awareness of the memory subsystem in the transformation process limits the applicability to x86 CPUs only. As important remark, it must be noticed that both works propose and implement an automatic flow to perform this software restructuring.

The hardware version, which as the name suggests is related to hardware accelerators design, consists of replicating the architecture demanded to perform one time-step a number of times putting them in cascade, *i.e.* the output of one architecture is the input of the next. This is the idea employed in this thesis, and is also proposed in [175], where the hardware accelerator is a composition of soft-processors that must be explicitly programmed, hence it is a totally different approach with respect to the one proposed in this thesis where an SST is an automatically derived microarchitecture, and in [94], where it is analyzed only from a theoretical point of view. Although the idea is already present in the state of the art, the work of this thesis is the first that propose a methodology to perform it automatically, along with some concepts, such as the already cited *queue looping*, which are completely novel.

6.8 Overview of the Methodology

We propose a methodology to derive an hardware accelerator designed to target a specific ISL. The hardware accelerator can be viewed, at every level of granularity, as a composition of *independent* modules, that communicate over FIFO channels and employ *blocking reads* and *writes* to manage the data flow and ensure its correct-

ness. In particular, we developed a **streaming architecture** aimed at performing a single ISL time-step, which we called SST (the entire accelerator is represented by the composition of multiple SSTs in a *queue fashion*). We then also proposed a *design automation flow*, to **automate** the SST derivation and the queuing process. This design automation flow is basically a 2-step process. The first step consists in deriving the microarchitecture that is demanded to implement a single iteration – *i.e.* a *time-step* – the SST. This can be viewed as the *basic building block* of the accelerator. The second part addresses how to actually building the complete accelerator, in which SSTs are arranged in a *queue* fashion, realizing an effective and explicit streaming, scalable mechanism. It takes as input the ISL's Static Control Part, written in an imperative form (*e.g.* in C/C++), and produces the corresponding accelerator. The proposed flow prepends to the aforementioned steps a *pre-processing* phase, in which the so called Reduced Static Control Parts (rSCoPs) are extracted. Both the preprocessing phase as well as the first macroblock heavily rely on the *polyhedral framework*, as most ISLs, due to their staticness and regular structure, can be viewed as a subset of the class of programs known as SANLPs. We restrict ourselves to treat only the ISLs enjoying this property. An overview of the proposed flow can be seen in Figure 6.6.

Let us briefly describe the two macroblocks of the proposed design automation flow. The first macroblock is the *SST microarchitecture derivation*, and is composed of the following parts:

- The first part performs the polyhedral analysis in order to extract a polyhedral IR of the input code, and also the corresponding Data Dependency Graph, which is crucial for the entire SST derivation process;
- The second part consists of an ad hoc manipulation of the obtained Data Dependency Graph in order to obtain the skeleton of the SST, which we call *streaming-oriented graph*;
- After that, two concurrent phases take place. Both phases employ the polyhedral IR along with the streaming-oriented graph, and their function is to further caracterize respectively the *memory system* and the *computing system* of the SST.

The result of this process is an IR of the derived SST, which is used to generate the code of the modules that will be synthesized via HLS.

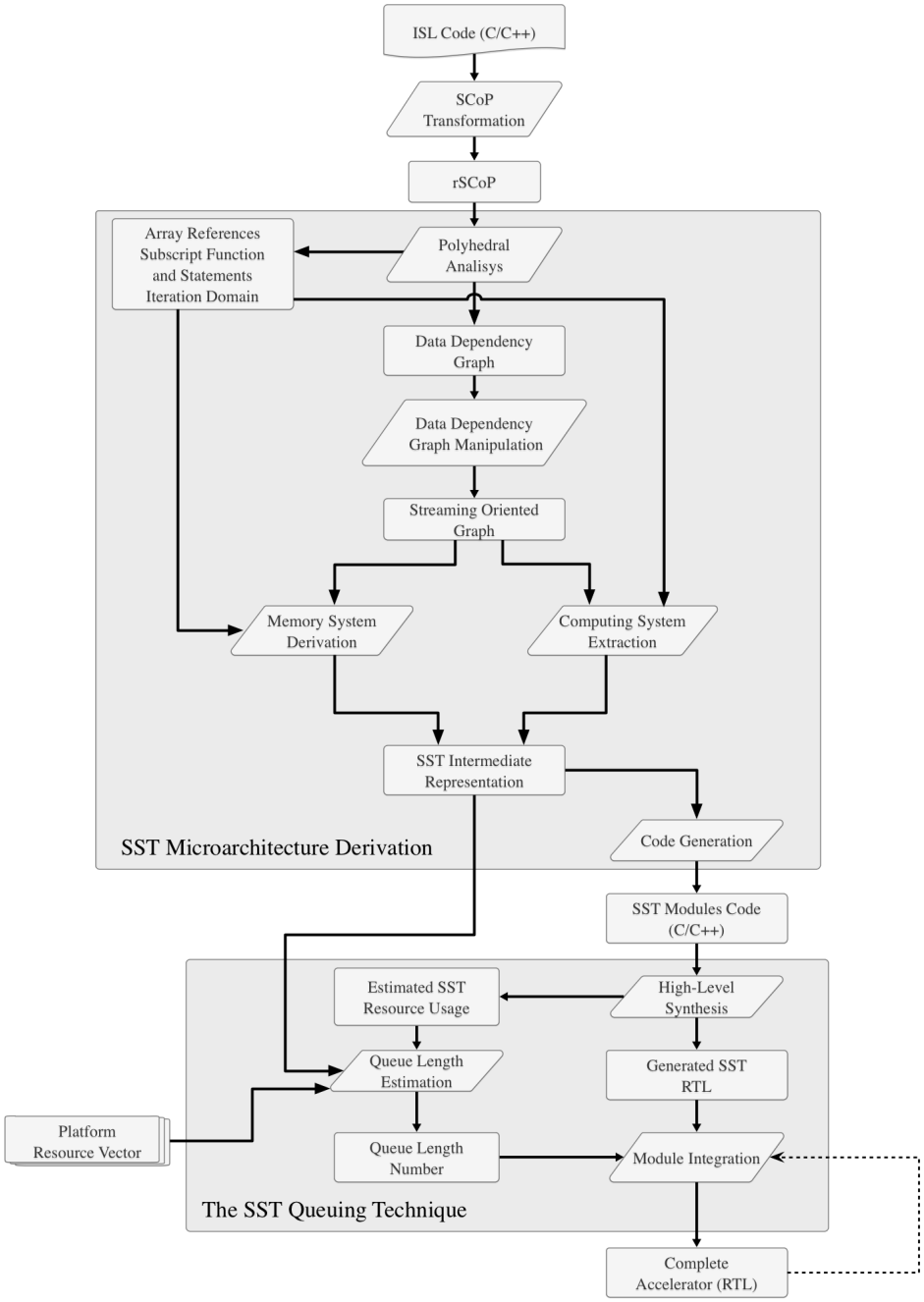


Table 6.6: *The Proposed Design Automation Flow. Each steps are described in Section 4 and 5.*

The second macroblock is the *SSTs queuing*, that employ the estimated resource usage of an SST and the total amount of available resources in order to derive the maximum achievable queue length and generate the final RTL of the resulting hardware accelerator.

As a final remark, it can happen that the ISL's SCoP contains conditionals which are *affine functions* of the outermost loop, that is indeed the time dimension. In this case, the code must be transformed beforehand, as a conditional on the time dimension means that only certain code parts are executed within a time-step, *i.e.* code parts execute in a mutually exclusive manner, and when deriving the SST, this situation is unacceptable. To deal with this case a solution is to apply the *index-set splitting* transformation along the outermost loop *only*. The affine conditionals can be used to effectively drive the splitting on the original loop nest, and after the transformation is performed it can be safely assumed that the conditionals can be removed from the obtained code. The result of this process is a series of loop nests, each one iterating over a subset of the original iteration vector of the time dimension. We call them rSCoP, since they actually still belong to a single SCoP, but despite this we treat each of them individually.

6.9 Proposed Architecture: the Streaming Stencil Time-step

A SST is demanded to execute a single ISL time-step and has *in general* one input stream and one output stream. In the case in which the ISL updates grid points employing constants or other arrays, the input streams are more than one. The components within an SST can be divided in two main categories, the first being the *memory system*, the second being the *computing system*.

The *Memory System* consists of a series of *chains* of modules connected via FIFO channels, one chain for every distinct input array, all responsible for feeding the computing architecture with the needed data. Each chain receives a single data stream – which is the array itself – and the modules within the chain represents the different read array references. These modules are the ones actually responsible of sending the data, as in fact they read any existing data element from their preceding FIFO and send the data element to the successive FIFO as well as to the computational system. From a high level perspective, this arrangement can still be viewed as a single stream, from were each module filters data only when needed, which is why

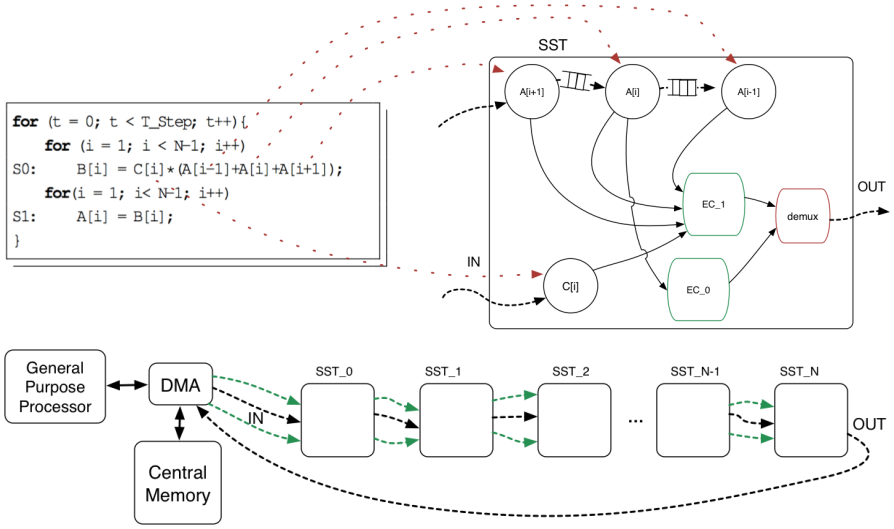


Table 6.7: *In Figure (a): an example of SST resulting from the sample code on the left. Each array relates and belong to a chain and each array reference corresponds to a new module inside this chain, as shown by the red arrow in figure. Each SST computes the code associated to a single, specific timestep. In Figure (b): The architectural template of the accelerator for an ISL with multiple inputs. The complete hardware accelerator is the composition of multiple SSTs as of Figure (a) arranged in a queue fashion. The green arrows represent the additional streams. As described in the text, the last SST has only the actual output stream.*

we called them *filters*. The chain-like organization of filters ensures that the data is read only once and at the same time allows more concurrent accesses, realizing also the optimal Full Buffering.

Computing System It is composed of a series of modules that perform the actual computation taking data from the memory system. Let us now provide some further details to better understand how they are arranged. Given the fact that in the case of ISLs there is always the presence of the boundary to take into account, there is the possibility of performance loss when hardware accelerators are employed, as the host processor could be forced to waste time to reconstruct the array from the output. For this reason, our SST consider the boundary in an explicit way. This is also a prerogative for the SSTs queuing. In fact, since the accelerator consists of a chain of replicas of a single SST, within an SST both the output and the input must be of the same form. To ensure that the output stream is rearranged in the exact same form of the input, we inserted a module – in addition to the modules that actually performs the computation – called *demux*, whose function is precisely the one just stated.

An SST has *in general* a single output stream. This is however not true when performing queuing with an ISL that takes multiple array as input (Figure 6.7). In fact, to provide the required data to all the SSTs within the queue, the SSTs are equipped with additional output streams, *i.e.* additional communication channels from the involved *filter chains* towards the next SST in the queue, other than the one that drives the proper accelerator's output. This is not true for the last SST, which does not have the added output streams as they are not needed. As shown in Figure 6.6, the first part of the proposed design automation flow is the derivation of the SST from the input rSCoP. Let us now describe the different phases that takes place within this macroblock.

6.9.1 Streaming-oriented Graph Construction

The purpose of this first phase is to construct a graph, namely *streaming-oriented graph*, which will be used as a skeleton for the SST. We use this approach as it can be automatically built on top of the already available tools of the polyhedral framework. Also, streaming architectures can be effectively represented as a graph, therefore a graph-like representation of the SST microarchitecture fits perfectly with its streaming nature – and eases the process of

information manipulation during the SST derivation. The first task that must be performed is the *data dependency analysis*, in order to produce, for a given rSCoP, the corresponding Data Dependency Graph. The only dependencies that must be taken into account are the RAW dependencies, as an SST can be viewed as an *in-order* microarchitecture. After that, the Data Dependency Graph must be further pruned: the dependencies carried by the time dimension – *i.e.* the outermost loop – *must* be discarded, since, as stated before, an SST is demanded to implement the execution of a *single* time-step. There is however a case in which an edge on the time dimension can remain after this pruning task. This situation can occur whenever the rSCoP loop nest is *imperfect*, because flow dependencies within the *same* time-step could be carried exactly along the time dimension. As a result, we can define the following conditions for an edge to be removed during the pruning task:

Definition 32. Data Dependency Graph Edge Removal Conditions. *Let us consider a Data Dependency Graph* $G = (N, E)$*, with only RAW dependencies, in which each node* N *is marked with a growing number given by the execution order of each statement, that, by the way, in the case of an rSCoP corresponds also to the* syntactic order*. For the process of the* streaming-oriented graph *construction, an edge* E *must be removed* if it represents a dependence carried along the outermost loop (time dimension, *and* E *is a self-loop or is directed from* N_i *to* N_j *and* $i > j$.

After that, *each* node (i.e., a statement) of the Data Dependency Graph is *expanded* in the following way:

1. Each array reference becomes a new node. Since for each array assignment – *i.e.* statement – the polyhedral analysis (indeed, the parsing step taking place at the beginning) is able to identify the read and write operations, we employ this information to connect these nodes. Specifically, each *read* node of the given assignment will have an outgoing edge connected to the corresponding *write* node, as the write operation trivially depends on these data. Furthermore, read nodes with the same array reference are merged. Note that the write nodes symbolically represent the execution statement, hence, they are associated with the statement's assignment – *i.e.* formula – and its iteration domain.

2. The original edges of the Data Dependency Graph are now

connected, rather than with the entire statement, to the specific node – *i.e.* array reference – involved in the corresponding dependence.

The last step towards the construction of a *streaming-oriented graph* consists in an iterative removal of the "copy" dependencies, *i.e.* an outgoing edge from the write node W of a statement entering another node N , such that:

- N has no outgoing edges which enters back W, causing a cycle;
- if N is a read node, the corresponding write node must not refer to the same array as W;
- if N is a read node, its *data domain* – *i.e.* the *image* of the corresponding statement iteration domain on the reference *subscript function* – matches the iteration domain of W;
- if N is a write node, its iteration domain is the same as of W.

Notice that this reduction can only be made if the aforementioned write node W does not have other outgoing edges. This operation is described by the pseudocode of Algorithm 10. The obtained graph

Algorithm 10: Iterative Reduction of the *Streaming-oriented* graph

Input: the streaming-oriented graph $G = (N, E)$
Output: the reduced version of G
$R = 0$; **foreach** *write node* $N \in G$ **do**
  if N *has only one outgoing edge* $E \wedge E$ *is directed to a node* N' *in a "copy" dependence relation* **then**
    $R \leftarrow R \cup (N, E, N')$;
  end
end
while $R \neq 0$ **do**
  remove $r = (N, E, N')$ from R; substitute the reference to N in N' with N assignment's formula; create a new write node W with the same formula of N'; substitute r in G with the single node W;
  if W *has only one outgoing edge* $E \wedge E$ *is directed to a node* W' *in a "copy" dependence relation* **then**
    $R \leftarrow R \cup (W, E, W')$
  end
end

is the skeleton of the SST microarchitecture, and we call it *streaming-oriented graph*. In order to have the IR of an SST, from this graph both the *computing system* and the *memory system* must be properly

characterized. In the following two sections we will explain the procedures to achieve this goal.

6.9.2 Computing System Extraction

The *streaming-oriented* graph provides us the write nodes that effectively become computation modules of the *computing system*. Since the streams entering the *filter chains* of the *memory system* must be in the form of the entire array, the boundaries must be explicitly managed. To do so:

1. Firstly, we make an important remark: the array updated within the statement with the highest index – *i.e.* the last in the syntactic order – must be the array updated by the ISL. We employ this information to explicitly identify the output array (stream) of an SST, as the one updated from the *last write node*;

2. A *demux* D is instantiated and associated to the *last write node* L such that L $\rightarrow$ D. How the boundary is actually managed is explained in 6.9.3;

3. The streaming oriented-graph is traversed, and each node N whose statement updates the same array of L, is also associated to D, *i.e.* N $\rightarrow$ D. If more than one write node is associated to the same demux, this means that there are different portions of the array which will be updated with different formulae. We call these portions *equivalence groups*. Since the write nodes are indeed responsible for the update of these regions of the array space, we will indistinctly refer to both regions and corresponding write nodes as *equivalence groups*. Let us now introduce the definition of an equivalence group.

   Definition 33. Equivalence Group. *An equivalence group is the maximal set* P *of points of a given array, updated within the ISL computation, such that each point of* P *has the same* update formula *and is dependent on the same set of* filters.

4. The process is repeated with all the remaining write nodes, whose array updates won't be forwarded as output of the SST. They will be only needed to ensure the correctness of the computation of an SST.

After the pruning task of the Data Dependency Graph previously described, the only kind of cyclic dependencies that can be present are statement's self-dependencies. When deriving the streaming oriented-graph, this translates into cycles between a given write node and some of its input read nodes, indicating the presence of spatial dependencies between grid points. In this case, the write nodes involved in these cyclic dependencies further require manipulation. From now on we will refer to the write nodes that enjoy this characteristic as *cyclic-write* nodes, and the read nodes which concur in the cyclic dependency as *cyclic-read* nodes. First of all, we claim that this cyclic dependencies can only involve read nodes whose *subscript function* is not of the form $f(\vec{x}) = I\vec{x}$ (being $f(\vec{x}) = F\vec{x} + \vec{a}$ where the *subscript matrix* F is the *identity matrix* I, and $\vec{a} = 0$). This condition is trivially enforced by the fact that these dependencies are between subsequent integral points of the iteration domain. Therefore a cyclic-read node *data space* (also called *data domain*) - *i.e.* the *image* of the cyclic-write node iteration domain on the reference *subscript function* - will always partially overlap with the boundary. Hence, since part of the data space of the cyclic-read node will come from the boundary and part from the cyclic-write node's output - *i.e.* two distinct data streams - the cyclic-read node will be actually implemented as two distinct filters, each belonging to a different chain. The result is that the iteration domain of each cyclic-write node is partitioned into subsets which are dependent on a different set of filters, even if the formula is actually the same. Hence, they are indeed different *equivalence groups*, as previously defined, even though we will refer to them as *sd-equivalence groups* (the prepended "sd" stands for *spatial dependence*), to differentiate them from the previously derived one. To partition the original cyclic-write node iteration domain, we need two basic information, namely the iteration domain of the sd-equivalence groups – each of them being a partition of the original iteration domain, and the set of input filters of each sd-equivalence group. The extraction of this information requires a specific algorithm, whose pseudocode is presented in Algorithm 11. An important precondition for the applicability of the algorithm is that the operations of *intersection* and *difference* between polyhedra can be performed by employing the state of the art library *isl* [201], a library for manipulating sets and relations of integer points bounded by linear constraints. The result is the set of sd-equivalence groups with the associated input filters and iteration

domain. Note that the read nodes which are not cyclic-read nodes are implicitly input of each sd-equivalence group, as they are indeed implemented as a single filter. The complexity of the proposed al-

Algorithm 11: sd-Equivalence Groups Extraction

Input: I: the iteration domain of the cyclic-write node w.
Input: A: set of cyclic-read nodes $a_i = (f_w, f_{nw})$, with a subscript function f_{a_i}. f_w represents the part of the data domain which overlaps with I, while f_{nw} represents the part which overlaps with the boundary.
Ouptup: E: set of equivalence groups $e = (i_e, r_e)$, where i_e is the iteration domain and r_e the set of input f_w.
$E \leftarrow 0$; $P \leftarrow 0$; {P is the set of the *preimage* portion of each a_i that overlaps with I}
foreach $a_i \in A$ **do**
  $D_{a_i} \leftarrow f_{a_i}$; $S \leftarrow D_{a,i} \cap I$; $p_{a_i} \leftarrow (f_{a_i}^{-1}(S),\ a_i)$ {the first element of the tuple is the preimage, the second is the identifier}; $P \leftarrow P \cup p_{a_i}$;
end
$i_0 \leftarrow \cap_i preImage(p_{a_i})$, $p_{a_i} \in P$; $E \leftarrow E \cup e_0 = (i_0, F = \{f_w(a_i) \mid \forall a_i \in A\})$;
$i_1 \leftarrow I - (\cup_i p_i,\ p_i \in P)$;
if $i_1 \neq 0$ **then**
  $E \leftarrow E \cup e_1 = (i_1, 0)$;
end
while $P \neq 0$ **do**
  $p_{a_j} \leftarrow firstElement(P)$; $Temp \leftarrow P - p_{a_j}$; $i_{new} \leftarrow preImage(p_{a_j})$;
  $r_{new} \leftarrow 0$; $r_{new} \leftarrow r_{new} \cup f_w(identifier(P_{a_i}))$;
  while $Temp \neq 0$ **do**
    $t_{a_k} \leftarrow firstElem(Temp)$; $Temp \leftarrow Temp - t_{a_k}$; $i_{old} \leftarrow i_{new}$;
    $i_{new} \leftarrow i_{new} \cap preImage(t_{a_k})$;
    if $i_{new} = 0$ **then**
      $i_{new} \leftarrow i_{old}$;
    else
      $r_{new} \leftarrow r_{new} \cup f_w(identifier(t_{a_k}))$;
    end
  end
  foreach $p_{a_i} \in P$ **do**
    $preImage(p_{a_i}) \leftarrow preImage(p_{a_i}) - i_{new}$;
    if $preImage(p_{a_i}) = 0$ **then**
      $P \leftarrow P - p_{a_i}$;
    end
  end
  $E \leftarrow E \cup e = (i_{new}, r_{new})$;
end

gorithm is $O(n^2)$ in the worst case, where n is the number of cyclic-read nodes. However, in real world cases, n is always small enough for the algorithm to terminate in a reasonable amount of time(*e.g.* in our benchmarks n has been no more than 4). It is also important to point out that this is, as far as we know, the first algorithm that en-

ables to automatically implement with an hardware accelerator ISLs with spatial dependencies between grid points.

6.9.3 Memory System Derivation

In order to characterize the memory system, the following steps have to be performed *for each write node*:

1. The cyclic-read nodes - whenever present - are implemented as two different filters, for the previously described reasons in subsection 6.9.2. The remaining read nodes will be implemented as a single filter. Filters are then clustered according to both the corresponding *array name* and the input stream - *e.g.* whether it is the output of the write node or not - to obtain the so called *chains*.

2. Whenever a chain contains more than one filter, those filters are ordered from the lexicographic maximum, to the lexicographic minimum. The input stream will enter the maximum flow following the *reverse lexicographic order*, down to the minimum, which means that those filters are linked together by the input stream. The sizes of the communication channels are computed as the modulus of the *data distance vector* of a *fixed* and *common* - but nevertheless arbitrary - iteration between the subscript functions of the two filters. Following the principles of [59], we can define the conditions for the proper structuring of a chain as:

   Definition 34. Chain Structuring Conditions. *A chain of ordered filters*
   $\{f_1 \rightarrow f_2 \rightarrow ... \rightarrow f_n\}$ *related to an array* A *must be compliant to the following two rules in order to have Full Buffering being also be deadlock-free:*

   - *For every couple* f_i *and* f_j *such that* $i < j$*, then*
     $$f_i \succ_l f_j$$
   - *The size* W *of a communication channel between a filter* f_i *with subscript function* f_A^i *and a filter* f_j *with subscript function* f_A^j *must be*
     $$W \geqslant \mid \delta(v, v)_{f_A^i f_A^j} \mid$$
     If W *is minimal* $(=)$*, the Full Buffering is also* optimal.

3. As stated before, there is also the boundary of the output array to consider. Within the chain referring to the same array updated by the write node (in the case in which the ISL has spatial dependencies the chain considered is the one *that does not take as input stream the write node's output*), the filter – whenever present – whose data domain perfectly overlaps with the iteration domain, *i.e.* for which the subscript function is $f(\vec{x}) = \mathrm{I}\vec{x}$ (intuitively, this is the "central" node of the chain), will be the one demanded to route the boundary towards the *demux*. If this node is not present (*i.e.* the update of a grid point is performed without reading its previous value), it will be added and its functionality will only be to route the boundary.

4. Each remaining communication channel, indeed every channel within an SST except the one inside the chains, will be of size 1.

6.9.4 SST IR and Code Generation

The purpose of the previous phases was to extract from the input rSCoP all the information needed to enable the actual implementation of an SST as an hardware microarchitecture. At the end of these phases, the information is encoded in the form of an IR. The IR contains:

- For each filter: an identifier; its data domain, which is the filtering condition; the input and output streams, *i.e.* the input and output communication channels
- For each equivalence group: an identifier; its iteration domain; the array update formula; the input communication channels, as well as the output one
- For each communication channel: an identifier; its minimum size

Informations about the demux are not needed as its structure is inferred from the iteration domain of the associated equivalence groups. From the SST's IR the hardware equivalent is generated employing HLS, hence, it is required to generate the code for each module of the microarchitecture: demuxes, equivalence groups and filters.The communication channels will be implemented as FIFOs queues. The

modules code can be generated using state of the art polyhedral model tools, integrated with the additional information we need in our case, such as read and write instructions on respectively input and output ports of the communication channels. Notice that the boundary transfer is actually made with a single channel from the involved filter to the demux. No additional modules will be inserted in between, as they are indeed unnecessary.

6.9.5 Pipelining the SST

Pipelining the equivalence groups can be an effective optimization to increase the overal throughput. However, since an SST is a composition of independent modules, this optimization could lead to situations of deadlock. To completely avoid this situation, we propose two solutions:

- The communication channels between the memory system and any equivalence group could be oversized, thus allowing the memory system to proceed even if the equivalence groups are stalled. However, to compute the communication channels size the pipeline depth of each equivalence group must be known, something that in general is not easy to do. Also, this solution will cause the SST to enjoy no more the optimal Full Buffering property.

- Only the innermost loop of each equivalence group is pipelined, which results into the flushing of the pipeline right when the memory system starts to send data to another equivalence group, completely avoiding the possibility of a deadlock within an SST.

There is a last, important, remark to be made. In the case of the presence of spatial dependencies, the related equivalence groups *cannot* be pipelined.

6.9.6 Scaling the Problem Size

Whenever the available on-chip memory is not large enough to allow the instantiation of all the communication channels, there is always the possibility to tackle the problem by trading bandwidth requirements for on-chip memory usage [59]. In practice, this means

that there is always the possibility to remove the *largest* communication channel and replace it with an additional input data stream from the off-chip memory. The process could be repeated iteratively until the overall memory requirements are compatible with the available resources. Notice that this trade-off possibility is however limited by the available bandwidth.

CHAPTER 7

Scaling Up: the SSTs Queuing Technique

In this Chapter, we build on the findings of the previous one in order to devise a mechanism to generate *chains* of hardware accelerators. By adding Streaming Stencil Time-steps (SSTs) to the chain, we linearly increase the overall throughput of the system while maximizing its overall power efficiency. Being streaming in nature, it is easy to delineate a mechanism to chain multiple accelerators together, forming what is subsequently called a *SST-queue*. The methodology allows to effectively scale-up the resulting computing system, as we show in the experimental part of this Chapter.

7.1 Introduction

The functionality of an SST is to perform the stencil computation associated to a single time-step. Hence, having an hardware accelerator built up of a lone SST would mean that, in order to perform more time-steps, the same SST should be employed over and over again, transferring back and forth data from the off-chip memory to the accelerator itself. These frequent off-chip memory transfers can

effectively bound the achievable performance, as an off-chip memory access is definitely much more expensive in terms of latency compared to data transfers within the accelerator. This is an issue already known in Iterative Stencil Loops (ISLs)' literature, as their inherent memory boundedness is a major reason due to which obtaining high performance codes and accelerators is in general an involved task. A possible solution to this problem could be to have a technique to limit as much as possible the off-chip memory transfers, exploiting the available hardware resources to offload not only the computation within a single time-step, but also the data transfers across time-steps. Our SST *queuing* technique goes exactly in this direction.

7.2 SST Queuing

The key point of the queuing technique is that multiple SSTs are arranged in a queue fashion, which means that, within the queue, the output of one SST is input to the next. Off-chip memory transfers occur *only* at the *beginning* and at the *end* of the queue. This implies that having a queue of arbitrary length or a single SST will involve the *same volume* of off-chip/on-chip memory transfers, which in turn means that the bandwidth requirements *will remain constant*, or more precisely, they are independent from the queue length. Therefore, the off-chip memory latency bottleneck is progressively alleviated as the queue length increase, since a greater volume of computation will be performed with the same off-chip bandwidth requirements.

In some occasions the SSTs data flow can be managed by an additional module, which is always aware of the progress of the computation as well as the total number of time-steps to be performed, which we called *mux*. This happens in two cases (which can also occur together):

- the number of SSTs – and hence the corresponding number of time-steps – of the hardware accelerator is *not* an exact divider of the total number of iterations. In this case the *mux* is responsible to break the computational flow when the total number of time-steps of the ISLs is performed, and redirect the output to the off-chip memory.
- the queue length is large enough to be able to cycle the data flow, redirecting the output stream of the queue back to the

queue itself, instead of transferring it back to the off-chip memory. We called this condition *queue looping*, which will be further detailed in section 7.4. In this case the *mux* is responsible to break this loop when the total number of time-steps has been executed.

It is important to note that the streaming nature of a single SST allows to have, at a certain point, all the SSTs concurrently processing on a "portion" of the stream, having therefore a *pipelined* computation within the queue. Also, the system is fully scalable, as the only constraint is the total amount of available resources. The fact that the achievable queue length is completely independent from the available bandwidth allows linear scalability of the size of the accelerator: the queue length can be enlarged until the available resources are saturated.

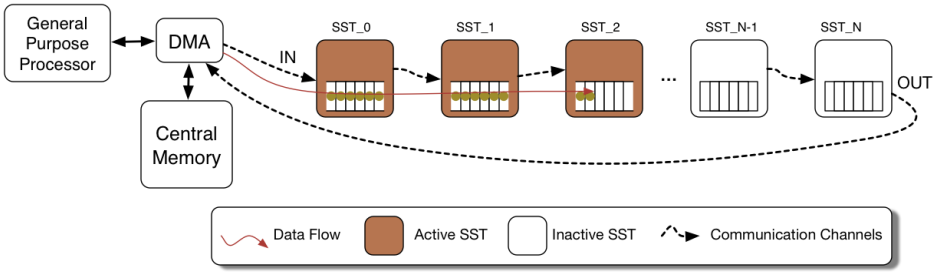


Table 7.1: *A visualization of the pipelined execution within the queue.*

Furthermore, we claim that employing the SSTs queuing will *speed up the Reduced Static Control Part (rSCoP) computation, and hence the throughput, by a pseudo-linear factor*, dependent on the number of SSTs instantiated within the hardware accelerator, *i.e.* the *queue length*. We provide a simple proof of this claim.

Proof. Let us model the completion time C of a given ISL when using an SST as hardware accelerator. We take as reference the state of a single grid point between two subsequent time-steps, thus:

$$C = T * (N * (m_{in} + sst + m_{out}))$$

where $T \in \mathbb{N}$ is the total number of time-steps, $N \in \mathbb{N}$ is the total number of points to be updated, $m_{in} \in \mathbb{N}$ is the number of clock cycles a given point takes to be transferred from the off-chip memory to the hardware accelerator, $sst \in \mathbb{N}$ is the number of clock cycles spent from an SST to actually update it, and $m_{out} \in \mathbb{N}$ is the number

of clock cycles it takes to be transferred back to the off-chip memory. Then, if we employ *queuing*, with a queue of length $q \in \mathbb{N}$, the completion time C_q is:

$$C_q = \frac{T}{q} * (N * (m_{in} + q * sst + m_{out}))$$

Whenever T or N are large numbers:

$$T * N \gg q \text{ and } T * N \gg m_{in} + sst + m_{out}$$

it would mean that:

$$T * N * (m_{in} + sst + m_{out}) \approx T * N * (m_{in} + q * sst + m_{out})$$

hence:

$$C_q \approx \frac{C}{q}$$

□

Remark 1. *The speedup is pseudo-linear because of the approximation* $T * N * (m_{in} + sst + m_{out}) \approx T * N * (m_{in} + q * sst + m_{out})$*, that is however a reliable approximation given the fact that* N *and* T *are very large in real ISLs.*

Queue Length Estimation

Within the proposed *design automation flow*, the queue length estimation is a process that takes as input:

- The estimated resource usage of an SST given from the High Level Synthesis (HLS), but also the resource usage of the communication channels, both platform dependent;
- The resource vector R_{max} which represents all the available resources;
- The total number t of time-steps of the rSCoP.

By employing this information, the estimation process is represented as simple division between R_{max} and the sum of the needed resources for both the communication channels and the SST, however limited in that the queue length q must be $q \leqslant t$. This limitation is nevertheless virtually nonexistent as in general ISLs are characterized by a very large number of time-steps. Interestingly, it should be noticed that there is an analytical bound to the queue length, and therefore a maximum number Q_{max} of iterations to be queued. When this analytical bound is reached, the stream can then flow back again in the queue instead of being transferred back to the off-chip memory, thus reaching the *maximum* achievable speed-up.

We call the condition for which the hardware accelerator is able to perform all the iterations of the ISL *queue looping*.

Definition 35. Q_{max} estimation. *An SST holds a fraction* f *of the sum of all the arrays involved in the computation, whose total size is* S_A*, hence:*

$$f = \frac{S_A}{k}$$

Therefore, the number of SSTs to be queued in order to perform queue looping *is:*

$$Q_{max} = \min\{q \mid \sum_q f > S_A\,,\ q \in \mathbb{N}\}$$

Lastly, we recall, as shown in Figure 6.6, that the actual implementation of the hardware accelerator *could be* an iterative process, since the estimated queue length may be too high to be able to instantiate the accelerator, whenever the available resources are not enough. This is indeed a platform related situation, as it depends on the accuracy of the resource estimation provided by the specific HLS tool. A simple solution could be to iteratively decrement the queue length (Figure 6.6) until the accelerator fits onto the available resources.

Handling More than One Input

The solution to handle more than one input array when performing the SSTs queuing has already been described in Section 32. The only detail that must be added is that in such a case, the HLS must produce two different versions of the SST: one with the added streams, and one with the only actual output. The filter within a chain demanded to forward the data will be the first. The way in which they are arranged is determined within the module integration phase.

7.3 Experimental Results

In order to test the proposed methodology and resulting hardware accelerator we select a number of significant ISL benchmarks, manipulate the code as described in Section 7.4, and generate the resulting system using Xilinx Vivado Design Suite. The SST architecture derivation has been aided by state of art polyhedral analysis tools. Specifically, we employ the following components:

- *Clan (Chunky Loop ANalyzer)*, to extract a polyhedral Intermediate Representation (IR) from the source code;

- *Candl (Chunky ANalyzer for Dependencies in Loops)*, to compute polyhedral dependencies, and thus the corresponding Data Dependency Graph, from the polyhedral IR.

The SST's modules have been implemented using *Vivado HLS* (v2014.3.1). Both SST's modules integration and queuing have been performed using the *Vivado* (v2014.3.1) toolchain, which is also employed to synthesize and implement the resulting RTL. Synthesis and implementation have been performed with an Intel Core i7-3630QM, featuring an 8GB DDR3 RAM. These specifications allowed us to push queuing only to a fraction of the total available resources, as we systematically ran out of memory during place and route with larger designs (i.e., with more SSTs enqueued). All the tests have been performed on a single VC707 board, which comes with a *Virtex-7* XC7VX485T Xilinx Field Programmable Gate Array (FPGA) chip. Along with other resources, the board features 1GB of DDR3 DRAM, which we also employed in our tests as reference off-chip memory.

7.3.1 Test Cases

Let us now focus on the benchmarks selected to validate both the methodology and the accelerator.

- *jacobi2D*: the fundamental computational kernel of 2D PDE solvers, and due to its nature it is well suited to perform SST queuing. As previously stated, our limited computational resources forced us to limit the queue length in order to successfully complete the accelerator synthesis and implementation. However, with *jacobi2D*, we were still able to push queuing up to a considerable number of SSTs without running out of memory during the synthesis process.
- *jacobi3D*: a 3-dimensional version of the above benchmark.
- *seidel2D*: this ISLs contains spatial dependencies between grid points updates, hence it has no trivial and/or explicit parallelization opportunities. This is usually the most complex kernel for automatic tools to analyze, and is relevant to test how well our methodology performs in the worst case (from the standpoint of data dependencies).
- *3D31pt*: a 31-point 3D, compute- and memory-intensive ISL with variable coefficients, and thus multiple input arrays. This

kernel is employed in different applications, and relates to 3D field solvers.

- *heat3D*: a discretized 3D heat equation stencil with non-periodic boundary conditions.
- *3D7pt*: a 3D 7-point stencil from the Berkeley auto-tuner framework.

All the benchmarks have been implemented using single precision *floating point* data types, both in hardware and Central Processing Unit (CPU).

7.3.2 Experimental Settings and Goals

We explored multiple operating frequencies during synthesis of the hardware accelerator. We could systematically synthesize without running in timing closure issues and/or out of memory exceptions at *200MHz*, which is our target frequency for the subsequent experiments. The datapath towards the off-chip memory is 32 bits wide, and the frequency 200MHz, so that the available bandwidth is 800 MB/s.

The experiments on the CPU side were conducted on an *Intel Xeon E5-1410*, a quad-core processor running at 2.8 GHz, with a peak performance of 179.2 GFLOPS. The benchmarks were compiled using Pluto [46] with diamond tiling activated, a state of the art technique for stencil optimization. For each of them we compiled both the original version and a diamond-tiled version running on 8 threads, where the optimal tile sizes have been determined empirically with a limited amount of search. Notice that for *seidel2D*, Pluto was not able to compile a diamond-tiled version, as its inherent sequentiality makes it not suitable parallelization-oriented optimizations.

Our goal is to demonstrate different aspects of our design:

- the efficient usage of the on-chip memory resources realized by an SST allows to treat problem sizes whose implementation would otherwise not be possible via direct synthesis or with trivial manipulation of the original source code via HLS,
- the scalability given by the SSTs queuing ensures a pseudo-linear increase in throughput, while keeping the off-chip bandwidth constant,

- improved power efficiency with respect to a general purpose processor, specifically when scaling out the design by queuing multiple SSTs.

7.3.3 Resource Usage

In this section we show the resource usage for all the benchmarks, expressed as a percentage of the total available resources (Figure 7.2). First of all, note how none of the benchmarks could neither be directly synthesized via HLS, nor synthesized without heavy hardware-oriented code restructuring (by also applying non trivial co-design considerations).

We have been able to synthesize successfully – i.e. without running out of memory during synthesis – 48 SSTs for *jacobi2D*. For *3D31pt* we could not synthesize more than 4 SSTs without running out of memory during the synthesis process on our machine, while for *3D7pt*, *jacobi3D* and *heat3D* we stopped at 8 SSTs for analogous reason. *Seidel2D* has spatial dependencies within points updates, thus a pipelined version cannot be obtained. Hence, only a no-pipeline version has been tested. We remark that in this case we were able to achieve, without running out of memory during synthesis, a queue length of 10 SSTs. For *3D31pt* we provide the floorplans of the 1 (in Figure 7.3) and 4 SSTs (in Figure 7.3) designs to visualize resource consumption and energy proportionality.

As we show in Figure 7.2, our methodology allows to consume an amount of resources proportional to the number of SSTs enqueued, with a correspondingly proportional increase in throughput. Thus, data clearly shows that our methodology is *energy-proportional*, a very desirable feature of systems, in general, and *heterogeneous systems*, in particular.

7.3.4 Performance

The performance of all the benchmarks are reported in Figure 7.5. Notice how, in all cases, the enlargement of the queue of SSTs corresponds to a pseudo-linear throughput speedup (as in Figure 7.6). Hence, data shows how our approach scales *by design*. It is also important to remark that the performance comparison between the proposed solution and the CPU must only be intended to provide at a glance an insight of the potential of our accelerator. As previously stated, the embedded system employed for validation was able to

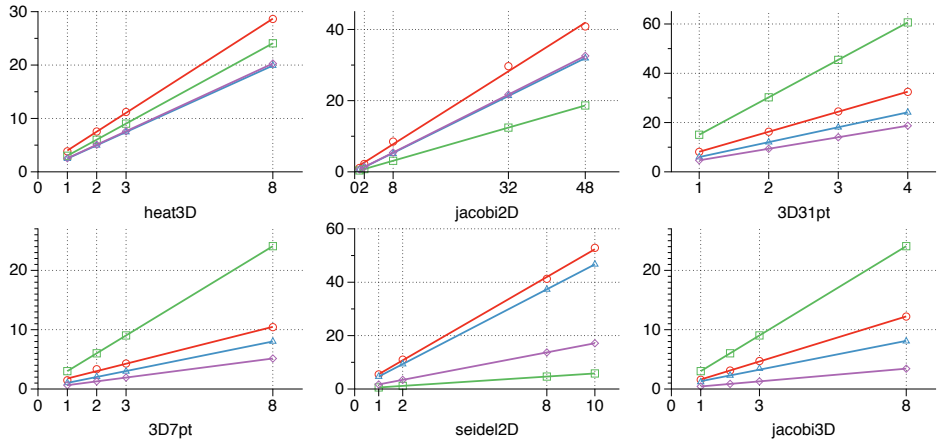


Table 7.2: *Resource usage of the accelerator for all the benchmarks. Legend: Flip Flops(FFs) (circles), Look Up Tables (LUTs) (triangles), Block RAMs (BRAMs) (squares), Digital Signal Processing Blocks (DSPs) (diamonds). Linear fits exhibit an* R^2 *coefficient always greater than* 0.99. *We recall that* R^2 *is the* coefficient of determination, *a coefficient ranging from* 0 *to* 1 *that indicates how well data fit the prediction model, which in this case is linear.*

deliver a maximum bandwidth of only 800MB/s, that is much less than the available bandwidth on the CPU. Future work will address this issue focusing on the design of a much more efficient underlying system, able to deliver sustained bandwidth and interface our hardware accelerator with a proper host processor – probably employing PCI-Express as communication bus.

7.3.5 Power Efficiency

Figure 7.7 shows the power efficiency normalized to the CPU. We measure power consumption at the wall, using a wattmeter. We claim that this methodology is sufficiently accurate for this experiment as the CPU spends most of the power on a compute intensive workload – CPU and RAM accesses, specifically. For all benchmarks, our final accelerator has a better power efficiency than the CPUs. This is in line with state-of-the-art works; however, our methodology allows to push the utilization of the target device, and thus to push the power efficiency to the limit imposed by the algorithm, device, and synthesis process (which all contribute to setting a hard limit to the power efficiency of a device). Data show that our

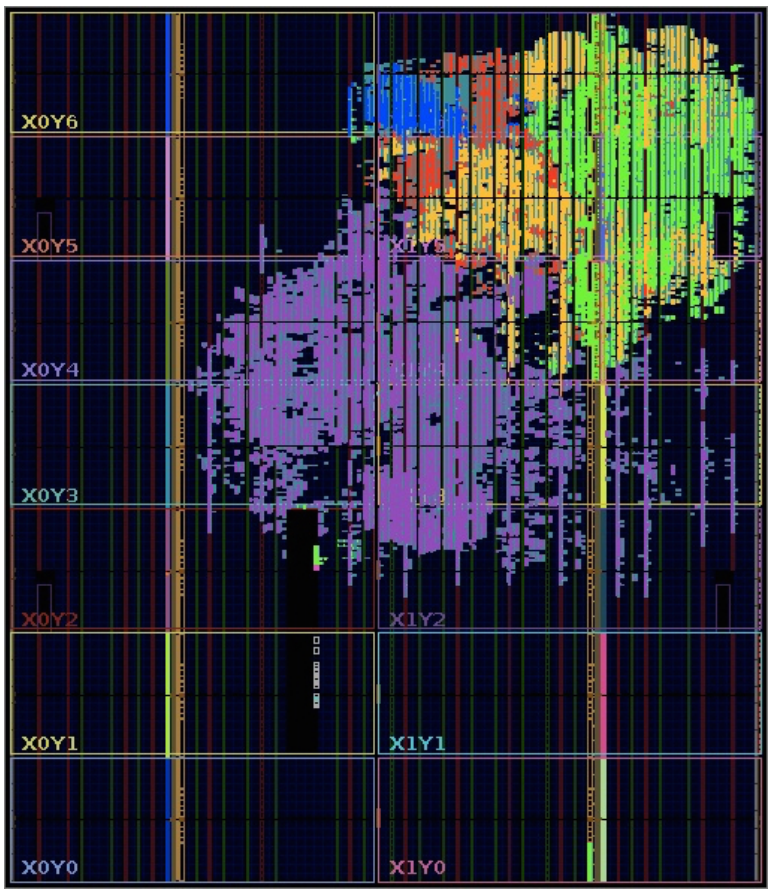


Table 7.3: *Floorplan of the 1 SST. In blue, the Microblaze used as host CPU to validate the accelerator. In green, the DDR3 controller. In red, the DMA. In ocher, the AXI interconnect of the design. In lily, violet, purple and pink, the various SSTs.*

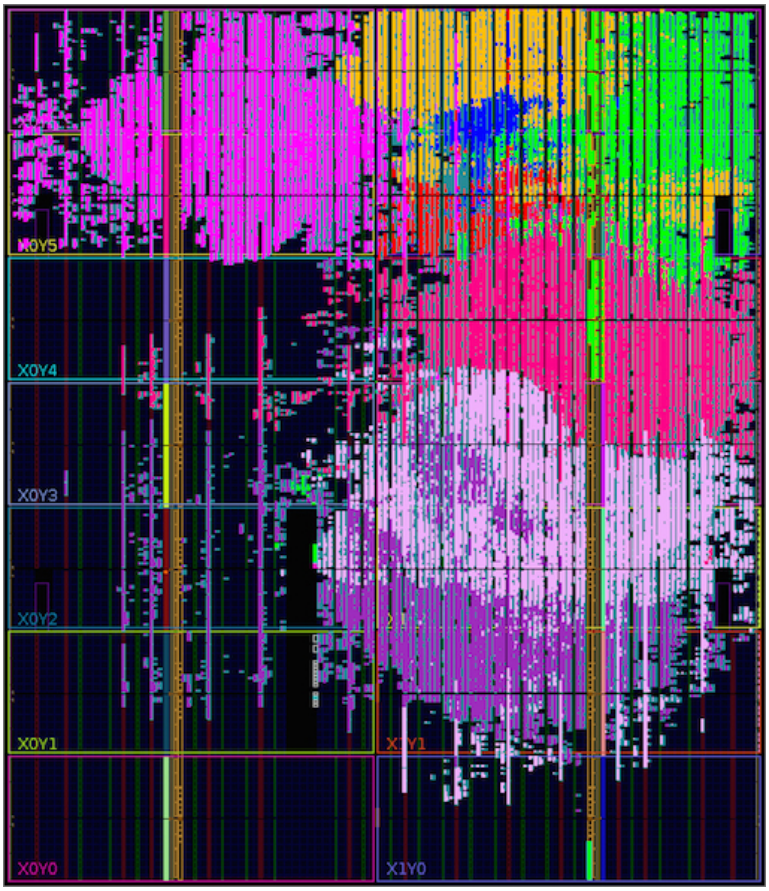


Table 7.4: *Floorplan of the 4 SSTs queued designs implementing* 3D31pt. *In blue, the Microblaze used as host CPU to validate the accelerator. In green, the DDR3 controller. In red, the DMA. In ocher, the AXI interconnect of the design. In lily, violet, purple and pink, the various SSTs. Observe how the relatively compactness of each accelerator allows the design to scale in resource linearly with each added IP.*

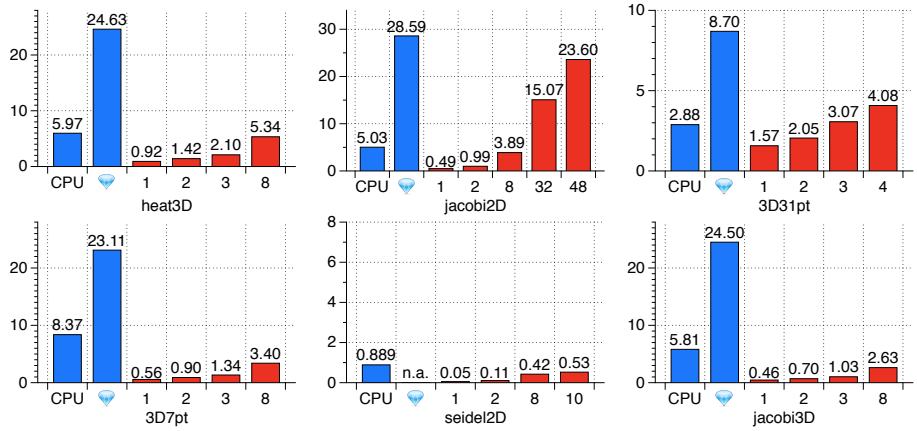


Table 7.5: *Throughput, in GFLOPS. CPU refers to the Intel Xeon E5-1410. "diamond" still refers to the throughput obtained on the Xeon, but running an 8-threaded version of the benchmarks, compiled using Pluto with diamond tiling activated. Finally, the other categories indicate the number of enqueued SSTs.*

methodology not only is *energy-proportional*, but it also allows the system to push the limits of the system's power efficiency, again a desirable feature when thinking about larger, scaled out systems.

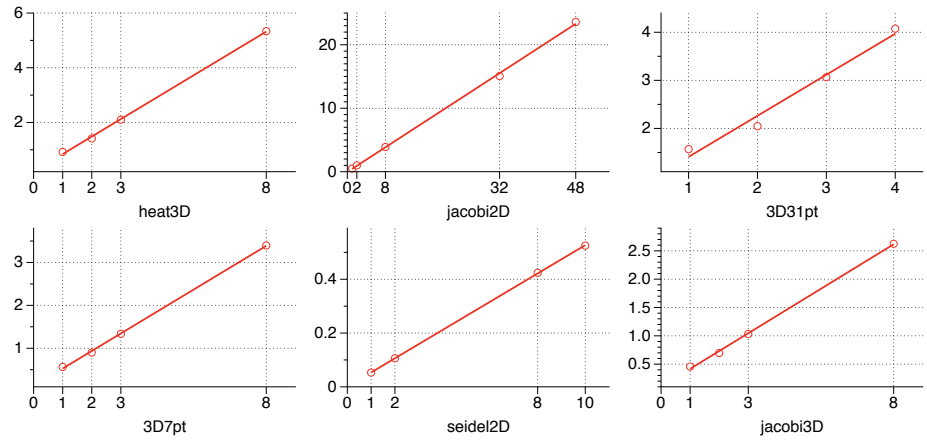


Table 7.6: *Throughput, in GFLOPS; hardware accelerators only. The number on the x-axis represents the number of enqueued SSTs. Also in this case, linear fits exhibit an* R^2 *coefficient always greater than* 0.99.

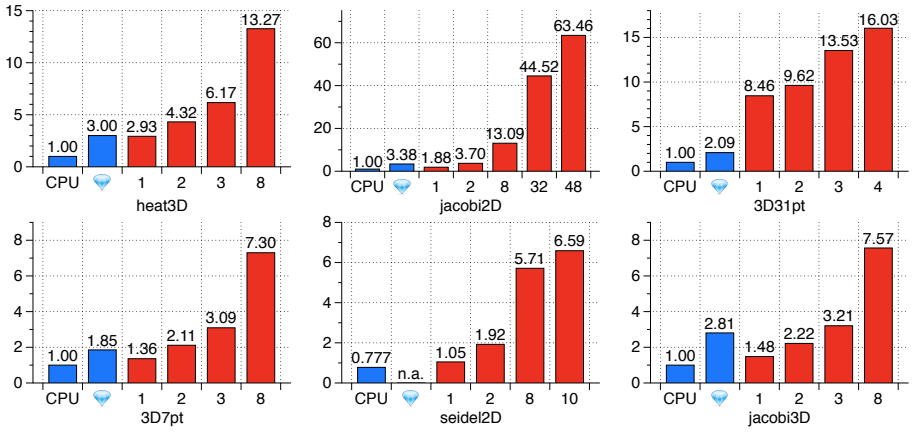


Table 7.7: *Power Efficiency ratio, normalized to the CPU. CPU refers to the Intel Xeon E5-1410. "diamond" still refers to the Xeon, but running an 8-threaded version of the benchmarks, compiled using Pluto with diamond tiling activated. Finally, the other categories indicate the number of enqueued SSTs.*

7.4 SST Generator

We developed a compiler that automates the generation of an SST-based architecture, *SST Generator*.

The methodology used to obtain the Streaming Stencil Time-step (SST) hardware accelerator is divided by three steps: extraction, intermediate representation and synthesis. During the *extraction* part, the pieces of code to accelerate in hardware are extracted from an source file written in C. This is done using the *clan* library, that implements the parser and produces a polyhedral representation of the relevant code (in our case, the code of the Iterative Stencil Loop (ISL)). After the extraction phase the code passes through two manipulation steps, namely Streaming Oriented Graph (SOG) and Streaming Oriented Architecture Graph (SOAG) generation. Finally, we translate the resulting data structure into hardware.

The first IR is obtained from the exact Data Dependency Graph (DDG) containing the Read After Write (RAW) dependencies of the previously extracted code. The DDG is computed using the *candl* library that analyzes the output obtained by the *clan* library. The first intermediate representation is the result of manipulations done on the DDG. The dependence arcs that represent a dependence over the temporal dimension of the ISL are removed. This is done because the relevant dependencies are the ones involved in the flow of data for a single timestep, so the dependencies along the time dimension are not relevant.

After the removal of the superfluous arcs, each node of the DDG is substituted with the matrix access that the respective statement performs. We call this new graph SOG. In a SOG each node represents an access to a matrix done by a statement. The access can either be a read or a write access, and each arc shows the flow of data between such accesses. All the read nodes are connected to their respective write node. The SOG is further simplified removing paths that do not alter data, whose only effect is delaying the propagation of that information (copy dependencies).

Afterwards, the SOG is further translated into the last intermediate representation, the SOAG. The SOAG stores all of the information required to build the actual hardware architecture. Each node in a SOAG models a hardware component that will be part of the final architecture, and each arc between two nodes models what will be a real wiring between hardware modules. This graph contains

	Throughput	Power Efficiency	FF	LUT	BRAM	DSP
	(GFLOPS)	(GFLOPS/W)	(%)	(%)	(%)	(%)
CPU + diamond tiling	20.458	0.455	-	-	-	-
8 SSTs	3.674	0.636	4.70	8.31	5.44	2.71
32 SSTs	14.345	2.299	9.81	15.48	10.34	10.86
48 SSTs	21.343	3.319	13.22	20.29	13.69	16.29
72 SSTs	31.204	4.649	18.33	27.48	18.72	24.43

Table 7.1: *jacobi2D.*

three kinds of nodes. First there are the *filters*, that memorize data from the data stream. They stream data to the components that perform kernel's actual computation. *FIFOs* accumulate these temporary data that will be used in future iterations/timesteps. Second, there is the computational *kernel*, which is the part that performs actual computation. Finally, there is the *demux*, a component in charge of reconstructing the output stream. During the construction of the SOAG, the required information are obtained using Polyhedral Analysis (PA). PA allows to decide the order the filters need to have to avoid deadlock in the final architecture, to compute the minimum size of each FIFO, and to tailor all the modules of the SST for the particular ISL taken into account.

There are few steps to perform in order to obtain a SOAG from a SOG. For each different matrix accessed in the ISL, an input channel needs to be instantiated. It will contain the modules filtering the streamed matrix to the kernel. Within each channel, the filters need to be ordered in inverse lexicographical in order to avoid deadlocks that will halt the architecture. Between each pair of communicating filters we instantiate a correctly sized FIFO. Their minimum size is computed through the analysis of the iteration domain associated to the ISL formula. Lastly, the demux is connected to the kernel and the input channel, allowing the correct reconstruction of the output matrix. The synthesis, using the information coming from the SOAG, produces a bitstream that can be used to program an Field Programmable Gate Array (FPGA) board. The SST itself is stored in a separate data structure in order to be able to build arbitrarily deep SST queues.

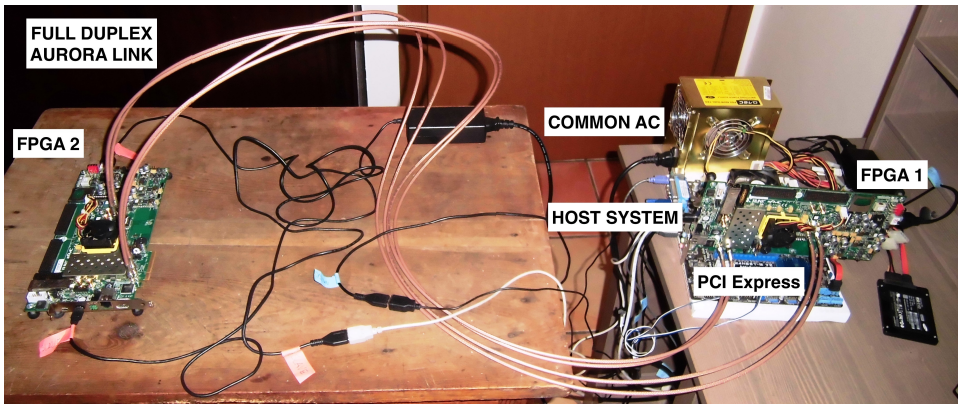


Table 7.8: *Experimental setup. Power absorption is measured at AC level, which is common to both host system, FPGA 1, and FPGA 2.*

	Throughput (GFLOPS)	Power Efficiency (GFLOPS/W)	FF (%)	LUT (%)	BRAM (%)	DSP (%)
CPU	0.910	0.020	-	-	-	-
8 SSTs	0.275	0.045	11.17	22.46	6.22	6.86
12 SSTs	0.411	0.065	15.25	30.71	7.38	10.29
16 SSTs	0.546	0.083	19.34	38.97	8.67	13.71
24 SSTs	0.813	0.117	27.52	55.47	11.24	20.57

Table 7.2: *seidel2D.*

7.5 Dual FPGA Results

In the experiments we connect together two FPGAs using a serial, point-to-point communication IP called Aurora, and instantiate on them an increasingly deep SST queue; we repeat this procedure for each benchmark. We collect data about throughput, logic occupancy, and power efficiency; the data are compared to the state of the art multicore CPU implementations in terms of throughput and power efficiency. We finally elaborate on the scalability of the approach.

7.5.1 Experimental Settings

The proposed hardware implementation has been generated using Vivado Design Suite 2015.2. We employ *SST Generator* to generate the Register-Transfer Level (RTL) and the IP cores for each SST module, starting from its C language specification.Synthesis and im-

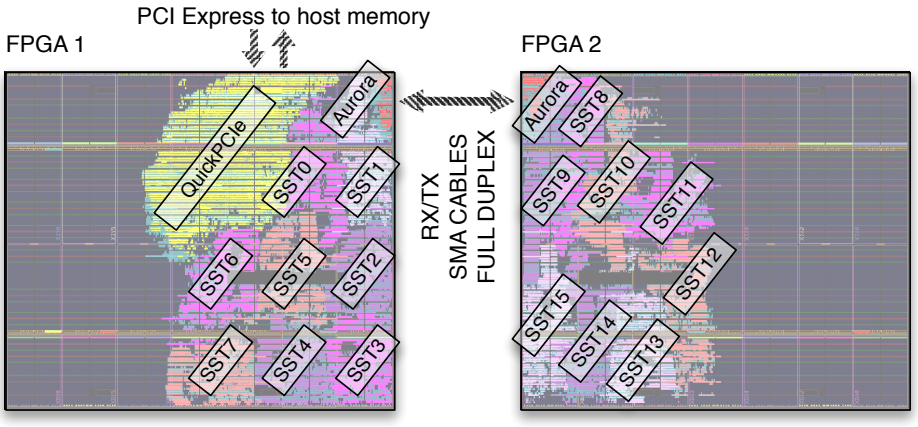


Table 7.9: *Experimental setup. Floorplan of the jacobi3D architecture on the multi-FPGA system with a SSTs queue of 16 SST instances (FPGA chip is XC7VX485T-2FFG1761C).*

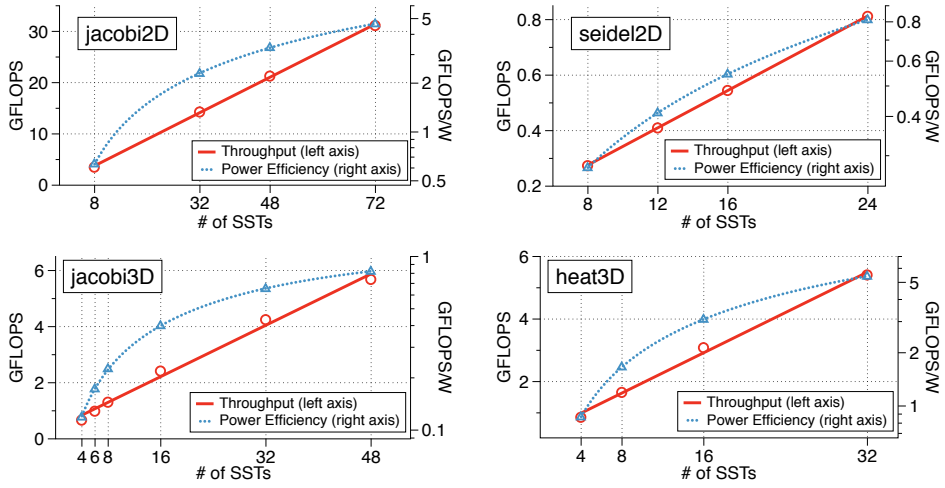


Table 7.10: *Log Power efficiency and throughput comparison between CPU and our implementation.*

plementation have been executed with an AMD Athlon II X4 640, featuring 8GB DDR3 RAM and running Ubuntu 14.04 x64. Due to the tight timing requirements for synthesis and implementation, and limited resources of the synthesis machine, we eventually ran out of synthesis machine RAM and *could exploit slightly more than half* of the available resources on any FPGA. All the tests have been per-

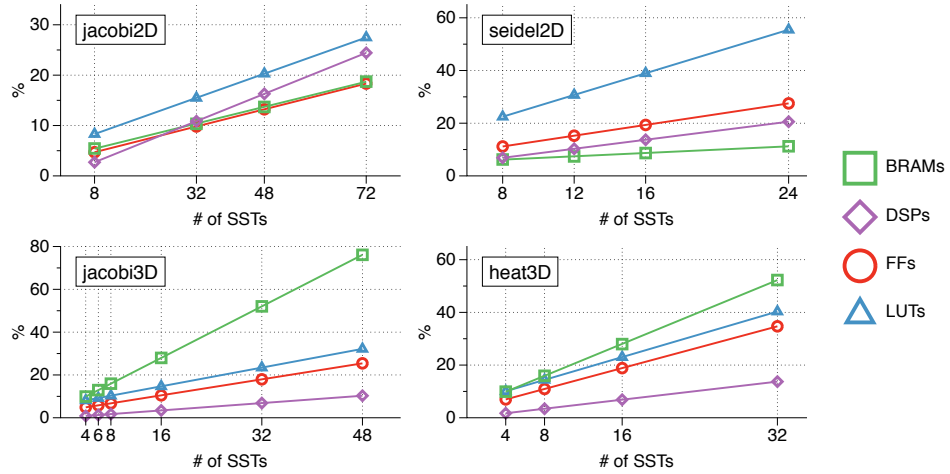


Table 7.11: *Resource consumption of increasingly lengthy SST-queues.*

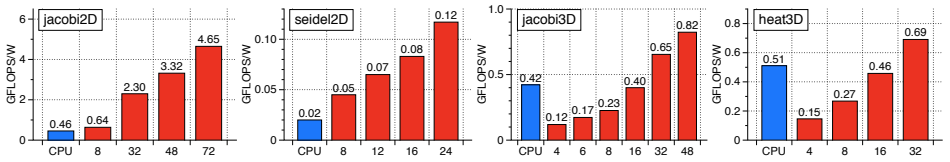


Table 7.12: *Power efficiency comparison between state-of-art CPU implementation (in blue) and our implementation (in red).*

formed with two *VC707* boards and with a host PC featuring an Intel Core i7 2675QM, 4GB DDR3 RAM, 256GB SSD running Ubuntu 14.04 x64. One of the two boards has been connected to the host PC through the PCI Express interface; the other was connected to the first through four coaxial cables, two for the TX channel and two for the RX channels, as per Aurora IP requirements (see Figure 7.8 and 7.9). In all tests we stream a fixed amount of data to the system; we measure the system when in "steady state" (*i.e.*: all SSTs are effectively computing on the data stream). Data are received by the first FPGA, processed by the SSTs therein, sent to the second through Aurora, further processed by the SSTs therein, and finally returned to the first FPGA where exchange with host memory happen, again via PCI Express. Tests have been executed with an increasing number of SSTs. Both the transfer of the data and the measurement of the performances have been managed by a custom application based on the provided *QuickPCIe* Intellectual Property (IP) core. We measure

	Throughput	Power Efficiency	FF	LUT	BRAM	DSP
	(GFLOPS)	(GFLOPS/W)	(%)	(%)	(%)	(%)
CPU + diamond tiling	18.982	0.422	-	-	-	-
4 SSTs	0.679	0.119	4.87	8.12	9.91	0.85
6 SSTs	0.998	0.173	5.81	9.22	12.92	1.28
8 SSTs	1.318	0.226	6.74	10.31	15.93	1.71
16 SSTs	2.429	0.400	10.48	14.69	27.96	3.43
32 SSTs	4.260	0.654	17.95	23.44	52.04	6.86
48 SSTs	5.695	0.823	25.42	32.18	76.12	10.29

Table 7.3: *jacobi3D.*

	Throughput	Power Efficiency	FF	LUT	BRAM	DSP
	(GFLOPS)	(GFLOPS/W)	(%)	(%)	(%)	(%)
CPU + diamond tiling	22.988	0.511	-	-	-	-
4 SSTs	0.869	0.146	6.97	10.06	9.91	1.72
8 SSTs	1.665	0.268	10.93	14.39	15.93	3.43
16 SSTs	3.094	0.458	18.85	23.02	27.96	6.86
32 SSTs	5.421	0.691	34.71	40.32	52.28	13.72

Table 7.4: *heat3D.*

power consumption at the Common AC power outlet.

7.5.2 Test Cases

The architecture has been validated against four benchmarks comprising relevant computational kernels in the linear algebra domain; their representativeness justifies their wide adoption in literature [28, 125]. These benchmarks are suitable for polyhedral-based optimisations and their structure permits to perform both SST generation and queuing. Additionally, their popularity allows us to fairly compare with other works, specifically with [46]. We run benchmarks activating diamond tiling on a more recent platform, that further improves the results obtained by the authors, as per data shown in Table 1 and Table 2. All the benchmarks have been tested with single precision *floating point* data types; the problem dimension for 2D benchmarks is 1000x1000; 100x100x100 for 3D ones. All matrices are dense. Total amount of FLOPS computation is computed by inspection of the statically defined loops.

7.5.3 Experimental Results

Referring to Figure 7.10, we observe how throughput increases proportionally with the length of the SST queue, which is consistent with the rationale behind the SST approach. Power efficiency tends to improve and then saturate with an increasing number of SSTs. This is a desirable feature as maximum FPGA utilization directly translates in maximum power efficiency. After a given SST-queue length, the system performs at full power efficiency while – theoretically speaking – arbitrarily increasing the throughput (by increasing the length of the queue). Additionally, we observe a linear increase in resource consumption, as reported in Figure 7.11; this is expected, as the computing system is the composition in a chain fashion of multiple copies of the same IP/SST. Observe how higher dimensional ISLs consume more local memory, and that seidel2D, which is more control-intensive then the other kernels, consume more LUTs to implement the custom control logic. These observations lead us to claim that our system is energy-proportional: resources, throughput, and power consumption proportionally grow together.

Let us now focus on the comparison with the optimal, state of the art multi core implementations. We regard as optimal our implementations as they are automatically parallelized using a locality-aware, parallelizing compiler adopting the diamond-tiling optimization strategy [46] (but for seidel2D, which is a benchmark not amenable to diamond-tiling). As far as we know, this is the best approach to simultaneous extraction of parallelism and locality for our workloads. As data show, after a given SST queue, power efficiency of proposed solution becomes better than CPU's; up to 10,11x in the case of seidel2D. We expect that longer SSTs queues translates into further improvements, as we see from Figure 7.10 that we haven't reached the power efficiency limit of our approach yet, with this number of SSTs. However, our synthesis machine cannot synthesize larger designs due to lack of main memory. Nonetheless, there is still margin for further improvements.

CHAPTER 8

Conclusions

In this thesis we introduced, explored, and partly addressed the problem of how to improve the power efficiency of reconfigurable hardware based computing systems, with a specific focus on high performance computing, polyhedral analysis, high level synthesis, and their inter relationships.

In the first part, we elaborated on the advantages of using Partial Reconfiguration (PR) to generate high performance reconfigurable hardware based accelerators. After demonstrating gains in terms of energy delay product and execution time, we introduced a mechanism to partition a set of hardware accelerators to distribute the workload among multiple computing elements, in the light of future multi-FPGA platforms.

Afterwards, we delineated how Polyhedral Model (PM) can be used to restructure the code to achieve better parallelization, obtaining as a final result an increase of the efficiency of hardware circuits. The proposed methodology faces the problem of creating an architecture that is suitable for the problem of accelerating the execution of data parallel codes when *tiles* are synthesized out of them. In

the experimental tests we show a slight increase in power consumption in the more parallel architecture respect to the basic High Level Synthesis (HLS) implementation, which is however largely compensated by speed ups ranging from 3x to 7x in overall execution time. In fact, the worst power consumption increase is in the order of 6% (about 100mW on our platform), a small term compared to the large gains in throughput which directly translate into better power efficiency. As explained in the previous Chapters, this methodology is effective only for intrinsically parallel algorithms and the additional limitation induced by the pureness of the input codes. As already explained, implementing synchronized access to shared data not only results in bottlenecks, but also a lot of effort should be put to design this features. However, as the focus is set on scientific algorithms this is a limited (if not irrelevant at all) issue, as most of those algorithms are easily expressible in this form.

Subsequently, we proposed a design automation flow to accelerate a target ISL on FPGA, consisting of a queue of architectures demanded to perform a single ISL time-step, the SST. We efficiently exploit the available resources realizing an optimal Full Buffering, and ensure a quasi-linear throughput speedup when enqueuing multiple SSTs, taking into account both memory and bandwidth considerations. We automatically derive the accelerator from the original source code, employing the polyhedral model in combination with HLS. Experimental results show an efficient usage of the on-chip memory resources realized by an SST, allowing to deal with problem sizes that would otherwise be untreatable with a direct synthesis of the original code via HLS. We also show that the SSTs queuing technique ensures a pseudo-linear increase in throughput obtained with constant bandwidth requirements. Also, the comparison shows that the proposed accelerator has the potential to outperform all comparable solutions thanks to its inherent scalability and specialization of the algorithm with respect to ISLs; overall power efficiency rivals the currently available top power efficient systems, and is expected to grow with the increase of the number of SSTs within the queue (i.e.: with the increase of logic utilization).

8.1 Future works

The first goal is to implement the final automatic toolchain. The second goal is to tighten the integration between task carried on by

difference researchers, like for example [57, 60, 62]. The third future goal will be to prove the feasibility of the multi-FPGA solution in order to implement memory intensive algorithms. Extensive tests on other computational kernels split on multiple FPGA will be required to demonstrate how performance scales with the number of computing elements. In Low-Level Virtual Machine (LLVM) related area, we already use Clang (via Chunky Loop ANalyzer (CLAN)) to translate code into PM. Till now we are using source-to-source transformation to create C file to feed Xilinx tools. Following the typical LLVM schema (front-end, IR, back-end) we could implements a different tool, that exploit directly the intermediate representation to generate the RTL of the circuits: in other words we could implement a typical LLVM back-end. The first step in this direction will be analyze and profile what has been already developed by LegUp project. Finally, the integration of this approach with large scale cluster software systems (like Hadoop) will prove beneficial in order to demonstrate the power efficiency figures in a more realistic computing system.

Bibliography

[1] A Free Framework for the High-Level Synthesis of Complex Applications. http://panda.dei.polimi.it/.

[2] Cuda home @ www.nvidia.com.

[3] Green500 List, November 2014. http://www.green500.org/lists/green201411.

[4] Green500 Power Efficiency Measurement Procedure, available at http://bit.ly/1MnTMEb.

[5] high-performance-xeon-phi-coprocessor-brief @ www.intel.com.

[6] index @ www.exascale-computing.eu.

[7] LegUp. http://legup.eecg.utoronto.ca/.

[8] mantle @ www.amd.com.

[9] Maxeler MaxCompiler. http://www.maxeler.com/products/software/maxcompiler/.

[10] Partial Reconfiguration Cost Calculator (PRCC), available at http://users.isc.tuc.gr/ kpapadimitriou/prcc.html .

[11] physx @ www.geforce.com.

[12] PoCC (Polyhedral Compiler Collection). http://www.cs.ucla.edu/~pouchet/software/pocc/.

[13] sdaccel @ www.xilinx.com.

[14] The Human Brain Project. https://www.humanbrainproject.eu/.

[15] Xilinx Vivado Design Suite, available at http://www.xilinx.com.

[16] List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321 – 1344, 1993.

[17] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012.

[18] The polyhedral compiler collection, 2013.

[19] A.V. Aho. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

[20] Mohammad H Al-towaiq. Parallel Implementation of the Gauss-Seidel Algorithm on k -Ary n -Cube Machine. 2013(January):177–182, 2013.

[21] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. FPGA-specific Synthesis of Loop-nests with Pipelined Computational Cores. *Microprocess. Microsyst.*, 36(8):606–619, November 2012.

[22] Corinne Ancourt and François Irigoin. Scanning Polyhedra with DO Loops. *SIGPLAN Not.*, 26(7):39–50, April 1991.

[23] F. Arandiga, A. Cohen, R. Donat, and B. Matei. Edge detection insensitive to changes of illumination in the image. *Image and Vision Computing*, 28(4):553 – 562, 2010.

[24] Pawan Kumar Aurora. Mmulti-level graph partitioning, 2007.

[25] Avnet Design Services ZedBoard. http://www.zedboard.org.

[26] C. Baaij, Matthijs Kooijman, J. Kuper, A. Boeijink, and Marco Gerards. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 714–721, Sept 2010.

[27] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. CHISEL: Constructing Hardware in a Scala Embedded Language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[28] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling Stencil Computations to Maximize Parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[29] S. Banerjee, E. Bozorgzadeh, and N.D. Dutt. Integrating Physical Constraints in HW-SW Partitioning for Architectures With Partial Dynamic Reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(11):1189–1202, 2006.

[30] S. Barnard. Pmrsb: Parallel multilevel recursive spectral bisection. *In Proc. Supercomputing*, 1995.

[31] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *In Proceedings of the* 6^{th} *SIAM Conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[32] Richard F Barrett, Shekhar Borkar, Sudip S Dosanjh, Simon D Hammond, Michael A Heroux, X Sharon Hu, Justin Luitjens, Steven G Parker, John Shalf, and Li Tang. On the Role of Co-Design in High Performance Computing. *vol*, 24:141–155, 2013.

[33] C. Bastoul. Extracting polyhedral representation from high level languages. Technical report, LRI, Paris-Sud University, 2008. Related to the Clan tool.

[34] Cédric Bastoul. Efficient Code Generation for Automatic Parallelization and Optimization. In *Proceedings of the Second International Conference on Parallel and Distributed Computing*, ISPDC'03, pages 23–30, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Cédric Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, Université Pierre e Marie Curie, December 2004.

[37] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, Olivier Temam, A Group, and Inria Rocquencourt. Putting Polyhedral Loop Transformations to Work. In *In Workshop on Languages and Compilers for Parallel Computing (LCPC'03), LNCS*, pages 209–225, 2003.

[38] Mohamed-walid Benabderrahmane and Albert Cohen. The Polyhedral Model Is More Widely Applicable Than You Think. *CC'10/ETAPS'10 Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Constructio*, 2010.

[39] Marsha J. Berger and Joseph E. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. Technical report, Stanford, CA, USA, 1983.

[40] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[41] A.J. Bernstein. Analysis of Programs for Parallel Processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757–763, Oct 1966.

[42] C. Bichot and P. Siarry. *Graph Partitioning*. 2011.

[43] Rainer Bleck, Claes Rooth, Dingming Hu, and Linda T. Smith. Salinity-driven Thermocline Transients in a Wind- and Thermohaline-forced Isopycnic Coordinate Model of the North Atlantic. *J. Phys. Oceanogr.*, 22(12):1486–1505, December 1992.

[44] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[45] Cristiana Bolchini, Antonio Miele, and Chiara Sandionigi. Automated Resource-Aware Floorplanning of Reconfigurable Areas in Partially-Reconfigurable FPGA Systems. In *In Proceedings of FPL '11*, pages 532–538, 2011.

[46] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, page 101, 2008.

[47] R. B. Boppana. Eigenvalues and graph bisection , an average-case analysis (extended abstract). *In Proceedings of the* 28^{th} *Symposium on Foundations of Computer Science*, pages 280–285, 1987.

[48] T. N. Bui and B. R. Moon. Genetic algorithm and graph partitioning. *IEEE Trans-actions on Computers*, 45(7), July 1996.

[49] T. N. Bui and L. C. Strite. An ant system algorithm for graph bisection. *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '02), W. B. Langdon et al. (Eds.)*, pages 43–51, 2002.

[50] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[51] Bing-Yang Cao and Ruo-Yu Dong. Nonequilibrium Molecular Dynamics Simulation of Shear Viscosity by a Uniform Momentum Source-and-sink Scheme. *J. Comput. Phys.*, 231(16):5306–5316, June 2012.

[52] A. Chambolle. An algorithm for total variation minimization and applications. *Journal of Mathematical Imaging and Vision*, 20:89–97, 2004.

[53] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. *2008 Symposium on Application Specific Processors*, pages 101–107, June 2008.

[54] Tatenda M. Chipeperekwa. Caracal: unrolling memory bound stencils. Technical report, - San Diego, La Jolla CA, USA, 2013.

[55] Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.

[56] J.A. Clemente, V. Rana, D. Sciuto, I. Beretta, and D. Atienza. A hybrid mapping-scheduling technique for dynamically reconfigurable hardware. In *in Proceedings of FPL '11*, pages 177–180, 2011.

[57] J. Cong. Behavior and communication co-optimization for systems with sequential communication media. *2006 43rd ACM/IEEE Design Automation Conference*, pages 675–678, 2006.

[58] Jason Cong, Muhuan Huang, and Yi Zou. Accelerating Fluid Registration Algorithm on Multi-FPGA Platforms. *2011 21st International Conference on Field Programmable Logic and Applications*, pages 50–57, September 2011.

[59] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 77:1–77:6, New York, NY, USA, 2014. ACM.

[60] Jason Cong, Stephen Neuendorffer, Juanjo Noguera, and Kees Vissers. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[61] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *Artificial Neural Networks and Machine Learning, ICANN 2014*, Lecture Notes in Computer Science. 2014.

[62] Jason Cong, Peng Zhang, and Yi Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, page 1233, 2012.

[63] Altera Corporation. Radar Processing : FPGAs or GPUs ? (May), 2013.

[64] P. Coussy, D.D. Gajski, M. Meredith, and a. Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, July 2009.

[65] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[66] Merkle D., Middendorf, M., and H. Schmeck. Ant colony optimization for resource-constrained project scheduling. In *IEEE Transactions on Evolutionary Computation*, volume 6, pages 333–346, 2002.

[67] S. Teng D. A. Spielman. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. *In Proceedings of the* 36^{th} *Annual ACM Symposium on Theory of Computing*, pages 81–90, 2004.

[68] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[69] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[70] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[71] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[72] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset. High-Level Synthesis of Loops Using the Polyhedral Model The MMAlpha Software.

[73] H. Ding and C. Shu. A Stencil Adaptive Algorithm for Finite Difference Solution of Incompressible Viscous Flows. *J. Comput. Phys.*, 214(1):397–420, May 2006.

[74] Marc Duranton, Daniel Black-Schaffer, Samia Yehia, and Koen De Bosschere. *The HiPEAC Vision*. 2011.

[75] R. Hassin E. M. Arkin. Graph partitions with minimum degree constraints. *Journal of Discrete Mathematics*, pages 55–65, August 1998.

[76] Johan Eker. Specification of the C AL actor language. 2003.

[77] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[78] H. Munk et al. Acotes project: Advanced compiler technologies for embedded streaming. *International Journal of Parallel Programming*, 2010. Special issue on European HiPEAC network of excellence members projects.

[79] Paul Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[80] Paul Feautrier. Dataflow Analysis of Scalar and Array References. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[81] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem: Part I. One-dimensional Time. *Int. J. Parallel Program.*, 21(5):313–348, October 1992.

[82] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[83] Paul Feautrier. *The Polytope Model: Past, Present, Future*. 2009.

[84] Christian Feichtinger, Johannes Habich, Harald Köstler, Georg Hager, Ulrich Rüde, and Gerhard Wellein. A flexible Patch-based lattice Boltzmann parallelization approach for heterogeneous GPU–CPU clusters. *Parallel Computing*, 37(9):536–549, September 2011.

[85] F. Ferrandi, P.L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 29(6):911–924, june 2010.

[86] Dietmar Fey. *Grid-Computing*. Springer, 2010.

[87] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partition. *in Proc.* 19^{th} *Design Automation Conf.*, pages 175–181, 1982.

[88] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[89] Alexandru Fiodorov. *Improving Energy Efficiency with Special-Purpose Accelerators*. PhD thesis, Norwegian University of Science and Technology, 2013.

[90] B. Fischer and J. Modersitzki. Fast Inversion of Matrices Arising in Image Processing. *Numerical Algorithms*, 22:1–11, 1999.

[91] P. O. Fjällström. Algorithms for graph partitioning : A survey. *Linköping Electronic Articles in Computer and Information Science*, 3, 1998.

[92] Matteo Frigo and Volker Strumpen. Cache Oblivious Stencil Computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[93] Matteo Frigo and Volker Strumpen. The Memory Behavior of Cache Oblivious Stencil Computations. *The Journal of Supercomputing*, 39(2):93–112, 2007.

[94] Haohuan Fu, Robert G Clapp, Oskar Mencer, and Oliver Pell. Accelerating 3D convolution using streaming architectures on FPGAs. In *SEG Expanded Abstracts*, volume 28, pages 3035–3039, 2009.

[95] S. Rao G. Even, J. Naor and B. Schieber. Fast approximate graph partitioning algorithms. *In SIAM Journal on Computing*, 28:639–648, July 1997.

[96] Dan Gajski, U C Irvine, and Irvine Ca. What Input-Language is the Best Choice for High Level Synthesis (HLS)? pages 857–858, 2010.

[97] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[98] J. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, pages 498–513, 1987.

[99] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, June 2006.

[100] D. Gohringer, M. Hubner, M. Benz, and J. Becker. A Design Methodology for Application Partitioning and Architecture Development of Reconfigurable Multiprocessor Systems-on-Chip. In *Proceedings of FCCM '10*, pages 259 –262, may 2010.

[101] T Grosser, A Größlinger, and C Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 2012.

[102] Robert M. Haralick and Linda G. Shapiro. *Computer and Robot Vision*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1992.

[103] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 95:452–469, 1995.

[104] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *In Proceedings of the ACM/IEEE Conference on Supercomputing*, page 626ñ657, December 1995.

[105] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[106] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[107] 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, pages 311–320, New York, NY, USA, 2012. ACM.

[108] Pao-Ann Hsiung, Marco D Santambrogio, and Chun-Hsian Huang. *Reconfigurable System Design and Verification*. CRC Press, 2009.

[109] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[110] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 36–43, New York, NY, USA, 2005. ACM.

[111] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The Organization of Computations for Uniform Recurrence Equations. *J. ACM*, 14(3):563–590, July 1967.

[112] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: application in vlsi domain. *Proceedings of the* 34^{th} *annual Design Automation Conference*, page 526ñ529, 1997.

[113] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Society for Industrial and Applied Mathematics Journal of Scientific Computing*, 20:359–392, 1998.

[114] Muhammad Umair Khan. Use multilevel graph partitioning scheme to solve traveling salesman problem. Master's thesis, Department Of Computer Engineering, Dalarna University, Sweden, 2010.

[115] M.S. Kim and K. Shimada. *Geometric Modeling and Processing - GMP 2006: 4th International Conference, GMP 2006, Pittsburgh, PA, USA, July 26-28, 2006, Proceedings*. Comuter-aided design. Springer, 2006.

[116] Ryohei Kobayashi. The 100-FPGA Stencil Computation Accelerator. Master's thesis, Tokyo Institute of Technology, Department of Computer Science, Graduate School of Information Science and Engineering, January 2013.

[117] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[118] C. C. Jay Kuo and Bernard C. Levy. Two-color Fourier Analysis of the Multigrid Method with Red-black Gauss-Seidel Smoothing. *Appl. Math. Comput.*, 29(1):69–87, February 1989.

[119] Y. M. Lam, J. Coutinho, W. Luk, and P.H.-W. Leong. Mapping and scheduling with task clustering for heterogeneous computing systems. In *Proceedings of FPL '08*, pages 275–280, 2008.

[120] Edward A Lee. Heterogeneous Concurrent Modeling and Design in Java (Volume 1 : Introduction to Ptolemy II). 1, 2008.

[121] Edward A Lee and Stephen Neuendorffer. Heterogeneous Concurrent Modeling and Design in Java (Volume 2 : Ptolemy II Software Architecture). 2, 2008.

[122] C. Lengauer, S. Apel, M. Bolten, A. Groblinger, F. Hanning, H. Kolster, U. Rude, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt. ExaStencils: Advanced Stencil-Code Engineering. In *Euro-Par 2014: Parallel Processing Workshops*, pages 553–564, 2014.

[123] P Li, L-N Pouchet, D Chen, and J Cong. Transformations for throughput optimization in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, 2014.

[124] Peng Li, Louis-Noël Pouchet, and Jason Cong. Throughput optimization for high-level synthesis using resource constraints. In *IMPACT*, 2014.

[125] Zhiyuan Li and Yonghong Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, November 2004.

[126] Xuejun Liang, Jack Jean, and Karen Tomko. Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems. *J. Supercomput.*, 19(1):77–91, May 2001.

[127] Christophe Lucarz and Marco Mattavelli. Dataflow / Actor-Oriented language for the design of complex signal processing systems. (Dasip), 2008.

[128] Christophe Lucarz, Marco Mattavelli, and Julien Dubois. A Platform for the Development and the Validation of HW IP Components Starting from Reference Software Specifications. *EURASIP Journal on Embedded Systems*, 2008(1):685139, 2008.

[129] J Marshall, A Adcroft, C Hill, L Perelman, and C Heisey. A finite-volume, incompressible Navier-Stokes model for studies of the ocean on parallel computers. *J. Geophys. Res*, (102):5733–5752, 1997.

[130] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979.

[131] Wim Meeus and Dirk Stroobandt. Automating data reuse in high-level synthesis. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 298:1–298:4, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[132] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An Overview of Today's High-Level Synthesis Tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[133] Richard Membarth, Frank Hannig, Jurgen Teich, and Harald Kostler. Towards Domain-Specific Computing for Stencil Codes in HPC. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1133–1138, November 2012.

[134] Jiayuan Meng and Kevin Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[135] Elad Mezuman and Yair Weiss. Globally optimizing graph partitioning problems using message passing. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, 2012.

[136] A. Morvan, S. Derrien, and P. Quinton. Efficient Nested Loop Pipelining in High Level Synthesis Using Polyhedral Bubble Insertion. In *Field-Programmable Technology*, FTP '11, pages 1–10. IEEE, 2011.

[137] Alessandro Antonio Nacci, Vincenzo Rana, Francesco Bruschi, Donatella Sciuto, Ivan Beretta, and David Atienza. A High-level Synthesis Flow for the Implementation of Iterative Stencil Loop Algorithms on FPGA Devices. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 52:1–52:6, New York, NY, USA, 2013. ACM.

[138] D. Nadezhkin, H. Nikolov, and T. Stefanov. Translating affine nested-loop programs with dynamic loop bounds into Polyhedral Process Networks. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 21–30, Oct 2010.

[139] Dmitry Nadezhkin. *Parallelizing Dynamic Sequential Programs using Polyhedral Process Networks*.

[140] Aiichiro Nakano, Rajiv K Kalia, and Priya Vashishta. Multiresolution Molecular Dynamics Algorithm for Realistic Materials Modeling on Parallel Computers. *Computer Physics Communications*, 83(2):197–214, 1994.

[141] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[142] G. Nicolescu, Ian O'Connor, and Christian Piguet. *Design technology for heterogeneous embedded systems*. Springer, 2012.

[143] H Nikolov, M Thompson, T Stefanov, A Pimentel, S Polstra, R Bose, C Zissulescu, and E Deprettere. Daedalus: Toward composable multimedia mpsoc design. In *Design Automation Conference*, DAC '08, 2008.

[144] Hristo Nikolov, Student Member, Todor Stefanov, and Ed Deprettere. Systematic and Automated Multiprocessor System. 27(3):542–555, 2008.

[145] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Multi-processor system design with ESPAM. *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis - CODES+ISSS '06*, page 211, 2006.

[146] Xinyu Niu, J.G.F. Coutinho, and W. Luk. A scalable design approach for stencil computation on reconfigurable clusters. In *Field Programmable Logic and Applications*, FPL '13, pages 1–4, 2013.

[147] Juanjo Noguera and Fernando Martinez Vallina. Zynq-7000 All Programmable SoC Accelerator for Floating-Point Matrix Multiplication using Vivado HLS. 1170, 2013.

[148] V. Osipov and P. Sanders. n-level graph partitioning. *In Proceedings of the* 18^{th} *European Conference on Algorithms: Part I*, 6346:278–289, 2010.

[149] Eunjung Park, Louis-Noel Pouche, John Cavazos, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Symposium on Code Generation and Optimization*, 2011.

[150] M. Peemen, A.A.A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *International Conference on Computer Design (ICCD)*, 2013.

[151] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11:430–452, 1990.

[152] Louis-Noël Pouchet. PolyBench/C the Polyhedral Benchmark suite.

[153] Louis-Noël Pouchet. *Interative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, Orsay, France, January 2010.

[154] Louis-Noël Pouchet. Polyhedral Compilation Foundations. Ohio State University. Course Lecture, 888.11, 2010.

[155] Louis-noël Pouchet, Cédric Bastoul, and Albert Cohen. LetSee : the LEgal Transformation SpacE Explorator.

[156] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[157] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *IEEE/ACM Fifth International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156, San Jose, California, March 2007. IEEE Computer Society press.

[158] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework . In *Conference on Supercomputing (SC'10)*, New Orleans, LA, November 2010. IEEE Computer Society Press.

[159] 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, pages 29–38, New York, NY, USA, 2013. ACM.

[160] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[161] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of Efficient Nested Loops from Polyhedra. *Int. J. Parallel Program.*, 28(5):469–498, October 2000.

[162] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1:95–113, 1989.

[163] Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. Understanding Stencil Code Performance on Multicore Architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 30:1–30:10, New York, NY, USA, 2011. ACM.

[164] Vincenzo Rana, Alessandro A Nacci, Ivan Beretta, Marco D Santambrogio, David Atienza, and Donatella Sciuto. Design Methods for Parallel Hardware Implementation of Multimedia Iterative Algorithms. (c):1–6, 2011.

[165] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 72:1–72:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[166] F. Rendl and H. Wolkowicz. A projection technique for partitioning the nodes of a graph. *Annals of Operations Research*, 58:155–179, 1995.

[167] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *IEEE Internal Symposium Parallel and Distributed Processing*, 2007.

[168] Lakshminarayanan Renganarayana, Manjukumar Harthikote-matha, Rinku Dewri, and Sanjay Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS*, 2007.

[169] F. Ritcher, M. Schmidt, and D. Fey. A Configurable VHDL Template for Parallelization of 3D Stencil Codes on FPGAs. In *European Regional Science Association Conference 2012*, ERSA '12, 2012.

[170] J.O.A. Robertsson. *Numerical Modeling of Seismic Wave Propagation: Gridded Two-way Wave-equation Methods*. SEG geophysics reprint series. Society of Exploration Geophysicists, the international society of applied geophysics, 2012.

[171] C. Roucairol and P. Hansen. Cut cost minimization in graph partitioning. *Numerical and Applied Mathematics*, pages 585–587, June 1989.

[172] Davod Khojasteh Salkuyeh. Generalized Jacobi and Gauss-Seidel Methods for Solving Linear System of Equations. *Numerical Mathematics, A Journal of Chinese Universities*, 16(2):164–170, 2007.

[173] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. SIAM.

[174] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi. Benefits and challenges for platform-based design. In *Proceedings of DAC '04*, pages 409–414, 2004.

[175] K. Sano, Y. Hatsuda, and S. Yamamoto. Scalable Streaming-Array of Simple Soft-Processors for Stencil Computations with Constant Memory-Bandwidth. In *Field-Programmable Custom Computing Machines*, FCCM '11, pages 234–241, 2011.

[176] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth. *Parallel and Distributed Systems, IEEE Transactions on*, 25(3):695–705, March 2014.

[177] Kentaro Sano, Satoru Yamamoto, and Yoshiaki Hatsuda. Domain-specific Programmable Design of Scalable Streaming-array for Power-efficient Stencil Computation. *SIGARCH Comput. Archit. News*, 39(4):44–49, December 2011.

[178] M.D. Santambrogio and D. Sciuto. Design methodology for partial dynamic reconfiguration: a new degree of freedom in the HW/SW codesign. In *Proceedings of IPDPS '08*, pages 1–8, 2008.

[179] N. Satofuka. *Computational Fluid Dynamics 2000: Proceedings of the First International Conference on Computational Fluid Dynamics, ICCFD, Kioto, Japan, 10-14 July 2000 \ Edited by Nobuyuki Satofuka*. Physics and astronomy online library. Springer Berlin Heidelberg, 2001.

[180] A. Schafer and D. Fey. High Performance Stencil Code Algorithms for GPGPUs. In *International Conference on Computational Science*, ICCS '11, pages 1–10, 2011.

[181] K. Schloegel, G. Karypis, , and V. Kumar. Graph partitioning for high performance scientific simulations. *In The Sourcebook of Parallel Computing*, pages 491–541, 2003.

[182] M. Shafiq, M. Pericàs, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguadè. Exploiting Memory Customization in FPGA for 3D Stencil Computations. In *Field-Programmable Technology*, FTP '09, pages 38–45, 2009.

[183] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.

[184] L.G. Shapiro and G.C. Stockman. *Computer Vision*. Prentice Hall, 2001.

[185] H.J. Siegel, Lee Wang, V.P. Roychowdhury, and Min Tan. Computing with heterogeneous parallel machines: advantages and challenges. In *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings., Second International Symposium on*, pages 368–374, Jun 1996.

[186] Scott Sirowy and Alessandro Forin. Where's the Beef? Why FPGAs Are So Fast. Technical Report MSR-TR-2008-130, Microsoft Research, September 2008.

[187] Gerard L. G. Sleijpen and Henk A. Van der. A Jacobi–Davidson Iteration Method for Linear Eigenvalue Problems. *SIAM Rev.*, 42(2):267–293, June 2000.

[188] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache Accurate Time Skewing in Iterative Stencil Computations. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 571–581. IEEE Computer Society, September 2011.

[189] Alejandro Fernández Suárez. Domain Specific Languages for High Performance Computing A Framework for Heterogeneous Architectures. pages 2012–2013, 2013.

[190] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. DeLite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[191] Summary Report of the Advanced Scientific Computing Advisory Committee Subcommittee. The Opportunities and Challenges of Exascale Computing. Technical report, U.S. Department of Energy, 2010.

[192] Synopsys, Inc. Platform Architect. http://www.synopsys.com/Systems/ArchitectureDesign.

[193] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[194] Richard Thavot, Romuald Mosqueron, Julien Dubois, and Marco Mattavelli. Hardware synthesis of complex standard interfaces using CAL dataflow descriptions.

[195] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MPSoCs. In *Proceedings of CODES+ISSS '07*, pages 9–14, 2007.

[196] Jan Treibig, Gerhard Wellein, and Georg Hager. Efficient Multicore-Aware Parallelization Strategies for Iterative Stencil Computations. *CoRR*, abs/1004.1741, 2010.

[197] Kuen Hung Tsoi and Wayne Luk. Axel : A Heterogeneous Cluster with FPGAs and GPUs. 2010.

[198] R. Krauthgamer U. Feige and K. Nissim. Approximating the minimum bisection size (extended abstract). *In Proceedings of the 32nd Annual ACM symposium on Theory of computing*, May 21-23 2000.

[199] S. van Haastregt and B. Kienhuis. Automated synthesis of streaming C applications to process networks in hardware. *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 890–893, April 2009.

[200] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral Code Generation in the Real World. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 185–201, Berlin, Heidelberg, 2006. Springer-Verlag.

[201] Sven Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[202] Sven Verdoolaege. Polyhedral Process Networks. In *Handbook of Signal Processing Systems*, pages 1335–1375. 2013.

[203] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: A Tool for Improved Derivation of Process Networks. *EURASIP Journal on Embedded Systems*, 2007:1–13, 2007.

[204] John A. Walington and V. Michael Jr Bove. Stream-Based Computing and Future Television. In *137th Society of Motion Picture & Television Engineers Technilac Conference*, SMPTE '95, pages 69–79, 1995.

[205] Robert A. Walker and Raul Camposano. Carnegie Mellon's (Second) CMU-DA System. In *A Survey of High-Level Synthesis Systems*, volume 135 of *The Springer International Series in Engineering and Computer Science*, pages 60–67. Springer US, 1991.

[206] Y Wang, P Li, P Zhang, C Zhang, and J Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Design Automation Conference*, 2013.

[207] Christian Weiß, Wolfgang Karl, Markus Kowarschik, and Ulrich Rüde. Memory Characteristics of Iterative Methods. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM.

[208] G. Wellein. Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization. In *Computer Software and Applications Conference*, COMPSAC '09, pages 579–586, 2009.

[209] R. Wester and J. Kuper. Deriving Stencil Hardware Accelerators from a Single Higher-Order Function. In *Communicating Process Architectures 2014*, CPA '14, 2014.

[210] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism.

[211] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[212] D. Wonnacott. Using Time Skewing to Eliminate Idle Time Due to Memory Bandwidth and Network Limitations. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, IPDPS '00, pages 171–, Washington, DC, USA, 2000. IEEE Computer Society.

[213] Xilinx Inc. Microblaze processor reference guide, 2012.

[214] Y. Yaacoby and P.R. Cappello. Scheduling a System of Affine Recurrence Equations onto a Systolic Array. *Proceedings of the International Conference on Systolic Arrays*, pages 373–382, 1988.

[215] J. Yan and P. Hsiao. A fuzzy clustering algorithm for graph bisection. *Information Processing Letter*, 52, December 1994.

[216] Tomofumi Yuki and Sanjay Rajopadhye. Parametrically Tiled Distributed Memory Parallelization of Polyhedral Programs. Technical report, Department of Computer Science, Colorado State University, Fort Collins, CO, June 2013.

[217] Yongpeng Zhang and Frank Mueller. Auto-Generation and Auto-tuning of 3D Stencil Codes on GPU Clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164, New York, NY, USA, 2012. ACM.

[218] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. Improving Polyhedral Code Generation for High-level Synthesis. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 15:1–15:10, Piscataway, NJ, USA, 2013. IEEE Press.

[219] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 9–18, New York, NY, USA, 2013. ACM.