POLITECNICO DI MILANO

Facoltà di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

On how to Accelerate Iterative Stencil Loops: A Scalable Streaming-based Approach

Relatore: Prof. Marco Domenico SANTAMBROGIO

Correlatore: Dott. Ing. Riccardo CATTANEO

Tesi di Laurea di:

Giuseppe Natale — Matricola n. 803783

Carlo Sicignano — Marticola n. 800774

Anno Accademico 2013–2014

This page intentionally left blank

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CLB	Control Logic Block
CPU	Central Processing Unit
CV	Computer Vision
DDG	Data Dependency Graph
DFE	Dataflow Engine
DMA	Direct Memory Access
DRAM	Dynamic RAM
DSE	Domain Space Exploration
DSL	Domain Specific Language
DSP	Digital Signal Processing
FB	Full Buffering
FIFO	First In First Out
FLOP	Floating Point Operation

FLOPS Floating Point Operations Per Second

FPGA Field Programmable Gate Array

GPGPU General Purpose Graphic Processing Unit

GPL General Purpose Language

HDL Hardware Description Language

HLS High Level Synthesis

HPC High Performance Computing

HPRC High Performance Reconfigurable Computer

ID Iteration Domain

ILP Integer Linear Programming

IOB Input-Output Block

IP Intellectual Property

IR Intermediate Representation

ISL Iterative Stencil Loop

ISPS Instruction Set Processor Specification

LUT Look-Up Table

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

OR Operations Research

PB Partial Buffering

PDE Partial Differential Equation

PE Processing Element

PM	Polyhedral Model
PDG	Polyhedral Dependency Graph
PRDG	Polyhedral Reduced Dependency Graph
RAM	Random Access Memory
RAR	Read After Read
RAW	Read After Write
RTL	Register-Transfer Level
RTM	Reverse Time Migration
rSCoP	Reduced Static Control Part
SANLP	Static Affine Nested Loop Program
SCoP	Static Control Part
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SoC	System-on-Chip
SRAM	Static RAM
SST	Streaming Stencil Time-step
VHDL	VHSIC Hardware Description Language
WAR	Write After Read
WAW	Write After Write

Summary

In *scientific computing* and in general in High Performance Computing (HPC), *stencil* computations play a crucial role as they appear in a variety of different fields of application, ranging from Partial Differential Equations (PDEs) solving, to computer simulation of particles interaction, to image processing and Computer Vision (CV), and a lot more. The computationally intensive nature of those algorithms has created the need of good solutions to efficiently implement them, in order to save both execution time, and energy consumption. This, in combination with their regular structure, has justified a wide study and the proposal of a lot of different approaches, in which virtually every kind of computing device currently available has been explored.

The work proposed in this thesis addresses Iterative Stencil Loops (ISLs) employing as enabling technology the Polyhedral Model (PM), with the aim of accelerate them using a Field Programmable Gate Array (FPGA) as target device. In particular, this research propose a *streaming-based microarchitecture* called Streaming Stencil Time-step (SST), able to achieve, thanks to an *optimal* Full Buffering (FB), a really low usage of the available resources as well as an efficient data reuse; and a technique, named SSTs *queuing*, able to effectively increase the throughput by a *pseudo-linear* factor, which exploits the characteristics of the proposed microarchitecture putting replicas of it in cascade, enabling, thanks to the streaming nature of the SSTs, a pipelined execution within the queue.

The methodology has been tested with some significant benchmarks on a *Virtex-7* using the *Xilinx Vivado* suite. Results show how 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 synthesizing directly the original code via High Level Synthesis (HLS), but also how the scalability given by the SSTs queuing ensure a pseudo-linear increase in throughput, while remaining with constant bandwidth.

Sommario

Nel vasto scenario della *scienza computazionale* e dell'High Performance Computing (HPC) in generale, le computazioni di tipo *stencil* giocano un ruolo fondamentale in quanto appaiono sistematicamente in una pletora di campi applicativi, spaziando dalla risoluzione di equazioni differenziali alle derivate parziali, alla simulazione dell'interazione di particelle, all'image processing e alla Computer Vision (CV), e molto altro. Data la loro natura computazionalmente pesante, nel tempo si è evidenziata la necessità di soluzioni implementative efficienti, con l'obiettivo di ridurre sia il tempo di esecuzione, che il consumo energetico. Questo, in aggiunta alla loro struttura regolare, ha giustificato un esteso studio ed una varietà di approcci proposti, in cui praticamente qualsiasi dispositivo di elaborazione attualmente disponibile è stato esplorato.

Il lavoro proposto in questa tesi si focalizza sulla implementazione dei codici stencil, definiti Iterative Stencil Loops (ISLs), utilizzando come tecnologia abilitante il Polyhedral Model (PM), con l'obiettivo di accelerarli su una FPGA. In particolare, questa ricerca propone una *microarchitettura streaming* chiamata Streaming Stencil Time-step (SST), capace di ottenere, realizzando un Full Buffering (FB) *ottimo*, un basso uso delle risorse disponibili ma anche un efficace riuso dei dati; ed una tecnica, chiamata *accodamento* delle SST, in grado di aumentare il throughput di un fattore *pseudo lineare*, e che consiste nello sfruttare opportunamente le caratteristiche della microarchitettura proposta collegandone in cascata delle repliche, abilitando, grazie alla natura streaming delle SST, un'esecuzione in pipeline all'interno della coda.

La metodologia è stata testata con alcuni significativi benchmark su una *Virtex-7* utilizzando la suite *Vivado* di *Xilinx*. I risultati mostrano come l'efficiente utilizzo delle risorse di memoria on-chip realizzato da una SST consenta di trattare problemi la cui dimensione non ne consentirebbe l'implementazione sintetizzando via High Level Synthesis (HLS) direttamente il codice originale, nonché come la scalabilità data dall'accodamento delle SST garantisca un incremento pseudo lineare del throughput, pur restando a banda costante.

1

Introduction

Anyone can build a fast CPU. The trick is to build a fast system.

– Seymour Cray

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 High Performance Computing (HPC) field and its evolution towards the Exascale era. 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 special attention on Field Programmable Gate Arrays (FPGAs), that current trends sees as a promising approach to meet the presented challenges. Finally, Section 1.4 provides a high level overview of the proposed work within the context of the transition towards Exascale computing.

1.1 Context

Over 3 millions cores, clustered into 16k nodes, where each node has 88 gigabytes of memory, for a grand total of over 1 petabyte, a power consumption of 24 megawatts (accounting also cooling) and a performance of 33.86 petaflops: this is *Tianhe-2*, today's top supercomputer [10]. Although from these numbers it is clear that HPC systems can now deliver performance whose order of magnitude was simply unimaginable at the time of the first supercomputers, there are still

certain classes of problems that are unmanageable with the currently available computing power. Therefore, it is time for HPC to take a step further.

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 Gigasacale (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, the systems will achieve Exascale (10^{18}).

The need of such a technology advancement can be justified with a simple claim: some of the key computational challenges, that are faced not only by industry, or science, but civilization as a whole, can be addressed thanks to Exascale computing. 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 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 [9].

Current trends, however, suggest that there is the need to explore alternative solutions, or the goal of achieving Exascale computing may remain only feasible on paper. 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 Exascale 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 probably cost more than 100 billion dollars, and require its own dedicated power plant and over 1 billion dollars per year to be powered [108].

- 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.

1.2 The Challenges of Exascale Computing

While designing a new system that can be competitive with HPC modern standards requires a non negligible effort, managing to make a performance leap of orders of magnitude is infinitely more complex. There are in fact some important challenges within the HPC field that must be addressed in the proper way in order to be able to make such an accomplishment [27]. 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, since it is absolutely critical to reduce its requirement of at least 2 orders of magnitude for future hardware and software technologies. 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 the best of breed system in terms of performance, the already cited *Tianhe-2*, 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.

The majority of the power consumed by supercomputers today is not used to handle computations, but is used to move data around the system. Indeed, we can model the power demand of the copper wires within a system as [101]:

$$\mathrm{Power} \approx \mathrm{B} \times \mathrm{I}^2/\mathrm{A}$$

Where B is the bandwidth of a wire, A is the cross-sectional area of the wire and I^2 is the length of the wire. From this model, which is a simple RC model that does not take into account all the variables that concur in the actual power consumption, it is already clear that:

- Power consumption increases proportionally to the bit-rate, so as we move to ultrahigh-bandwidth links, it can become a major concern;
- Power consumption is highly distance-dependent, as it is quadratic with the wire length.
- Making smaller-sized wires will not improve the energy efficiency or data carrying capacity.

Therefore the emerging constraints on energy consumption will effectively influence the way of designing an HPC systems, 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 energy consumption for an Exascale system to keep its operational cost 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* [5], 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 system would need an improvement of $10\times$ 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 undoubtedly related to information storage, 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, still remaining within an acceptable power demand.

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 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 enormous progress in this sense have still been made, 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.

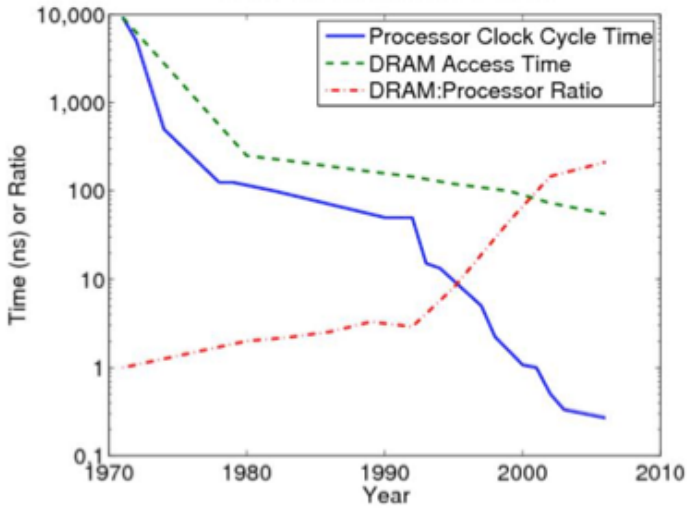


Figure 1.1: An illustration of the von Neumann Bottleneck. The graph refers to the evolution of canonical CPUs.

This problem is known as the "von Neumann bottleneck", as it can be thought as *structural*, related to how computation systems are made. 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 from Static RAM (SRAM). This type of memory is constructed from transistors and it has lower latency with respect to DRAMs, but it has very low data density and it is also more susceptible to errors that force them to be designed with error correction logic, resulting in higher costs, the reason why they are small-sized and

still need to be backed by traditional DRAM. Considering this hierarchical nature of memory, it is obvious that is preferable for the processor to find the piece of required data in the cache, as it can be accessed more rapidly. Hence, when the processor needs to read or write a location in main memory, it first checks if it is available in the cache. If the data is present this is called a cache hit, if not, it is a cache miss. When the cache misses ratio is high, the impact given by the employment of a hierarchical memory system can be completely null, therefore care must be taken in order to exploit it properly.

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 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 knows as *dark silicon* [45], caused by the failure of *Dennard scaling* [43], *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 force 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. The inability to go beyond a certain limit is indeed influencing also the employed parallelism paradigms, as in fact the pure many-core parallelism is being

gradually replaced by forms of process-level parallelism, an example of which is *MapReduce* [42].

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 more the system is large, 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.

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 is it 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* coexists, 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.

1.2.6 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 terms of comparison. Within the HPC field, the most significant metrics are:

- Throughput, measured in FLOPS,
- Bandwidth, measured in bit/s,
- Total Power and Power Efficiency, measured respectively in watts (W), and FLOPS/W.

1.3 Meeting the Challenges: 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 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 computational performance keeping low the energy requirements. Heterogeneous computing [103] 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 Central Processing Unit (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.

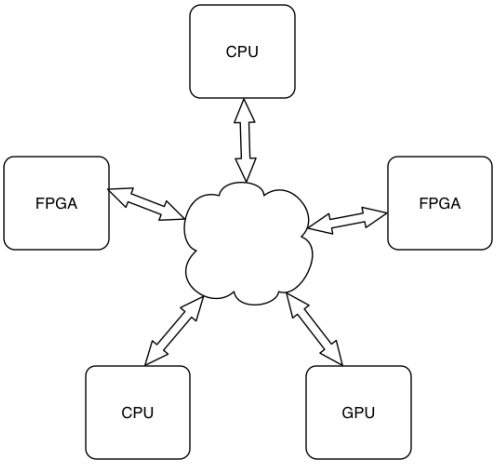


Figure 1.2: A simple scheme of an heterogeneous system.

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 [61]: the Control 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 Processings (DSPs) and other specific processors.

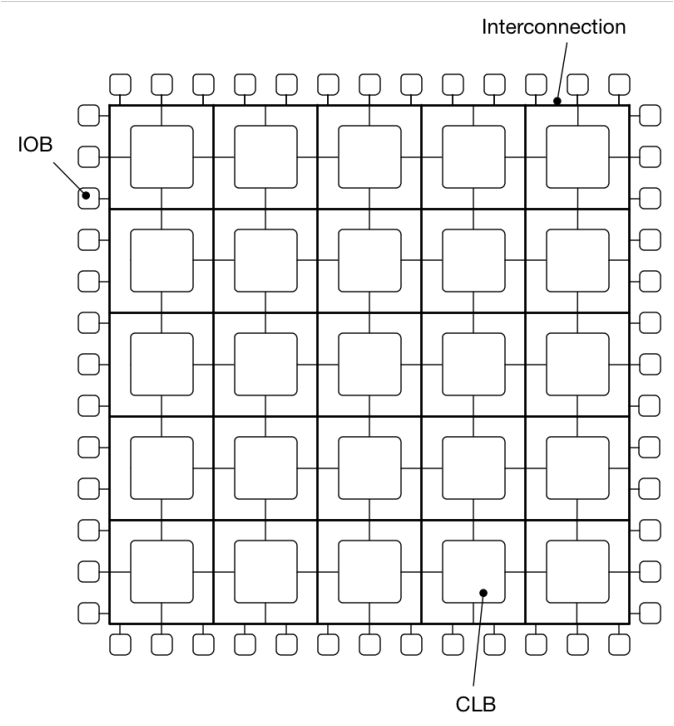


Figure 1.3: The general architecture of an FPGA.

The structure of an FPGA enables tasks-tailored logic to be created 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 [19].

1.4 A Contextualized Overview of the Proposed Work

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. We restrict ourselves to treat one class of algorithms, namely the Iterative Stencil Loop (ISL), a relatively small but very interesting domain in

the context of HPC, as they systematically appear in both scientific and industrial related computations. ISLs usually operate on multi-dimensional arrays, with each element computed as a function of some neighboring elements, and these neighbors represent the *stencil*. A thorough description of them will be provided in section 2.4.

In this work, we attempted to meet the presented challenges for the class of ISLs by designing a domain-tailored hardware accelerator, which relies on the following key points:

- the employment of custom logic to deliver *high power efficiency*;
- a custom memory architecture to *reduce data movements to the minimum*, realizing an efficient data reuse;
- a technique that realize *linear scaling* which enables throughput increase with constant bandwidth requirements;
- an inherent *parallelism* due to the use of a distributed microarchitecture, which fits perfectly with the distributed nature of an FPGA;
- a *design automation flow* to automatically derive the accelerator from the input source code.

1.5 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2 the needed background knowledge is provided, where all the basic concepts related to this work are presented. In Chapter 3 the thesis motivation is clearly stated, followed by a general overview of the proposed solution. Chapter 4 describes all the details related to the proposed design flow and the resulting hardware accelerator, whose evaluation is reported in Chapter 5. The conclusions are then presented in Chapter 6, along with some considerations on possible future work.

2

Background Knowledge

In order to easily read through the rest of this thesis, a full understanding of the background is absolutely essential. Hence, in this chapter the state of the art of technologies, methodologies and key concepts relevant to this work are thoroughly examined. In Section 2.1 the Polyhedral Model (PM) is presented, as it will be employed to provide automatic information extraction from the input program, while in Section 2.2 Streaming-based Systems on Field Programmable Gate Arrays (FPGAs) are introduced, since the proposed architecture is streaming-based as well. Section 2.3 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 2.4 focus is on Iterative Stencil Loops (ISLs), which are indeed the target of this entire work.

2.1 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 obviously imply 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 appro-

priate to perform such a task, as the the kind of abstraction of those techniques inevitably hides certain properties and features of programs, making 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 [89, 123], motivated by the need to map parallel computations onto *systolic arrays* [73]. It was based on the work of Karp et al. [63] that proposed a mathematical model which mapped onto uniform recurrence equations, which also inspired a series of fundamental papers from Feautrier [46, 47, 48, 49], arrived in the late eighties as well and quickly followed by other works related to the same topic, such as [119, 120, 15]. 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 [56, 13].

In the following section, a detailed overview of the polyhedral framework is then provided, starting from the model, described in section 2.1.1, and explaining what can be accomplished with such a model, which is the topic of section 2.1.2.

2.1.1 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 describe 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 [82].

Definition 2.1.1. *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 2.1.2. *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 2.1.3. *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 2.1.4. *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 2.1.5. *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 2.1.6. *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 2.1.7. *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 2.1.8. *Polytope*. A polytope is a bounded polyhedron.

Definition 2.1.9. *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 2.1.10. *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 2.1.11. *$\mathbb{Z}$-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 2.1.12. *Static Affine Nested Loop Program (SANLP)* [76]. 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, it the one of Static Control Parts (SCoPs). A SCoP is a subclass of general loops nests that can be represented in the polyhedral model [24].

Definition 2.1.13. *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 2.1.14. *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 2.1.15. *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 2.1.16. *Iteration Domain [48]*. 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 $\mathcal{D} = \{\vec{i} \,|\, \mathbf{P}\vec{i} \geqslant \vec{b}\}$

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]=...
```

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

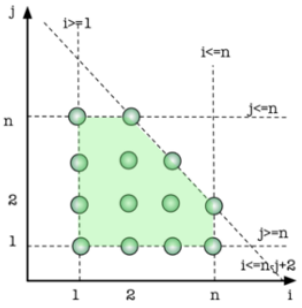


Figure 2.1: ID Example.

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 2.1.17. *Lexicografic Order* [48]. 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 2.1.18. *Subscript Function [20].* 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];
}
```

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

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

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

Figure 2.2: Subscript Function Example. The three subscript functions are relative to the three array accesses for s, a and x.

Definition 2.1.19. *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 2.1.20. *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 2.1.21. *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 iteration 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 2.1.22. *Lexicographically non-negative Distance Vector*. A distance vector v is lexicographically non-negative when the left-most entry in v is positive or all elements of v are zero.

Definition 2.1.23. *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 2.1.24. *Bernstein Conditions [28].* 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 [58]:

- **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. [82] 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 relation 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 2.1.25. *Dependency of statement instances*. A statement S depends on a statement R ($\mathsf{R} \rightarrow \mathsf{S}$), 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 belongs to the ID of R and S
- in the original sequential order, $S(\vec{x}_S)$ is executed after $R(\vec{x}_\mathsf{R})$.

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

Definition 2.1.26. *Data Dependency Graph*. A Data Dependency Graph (DDG) $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ is a directed multi-graph with each vertex representing a statement. An edge $\mathsf{e} \in \mathsf{E}$, from R to S represent 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 2.1.18). 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 2.1.27. *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:

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

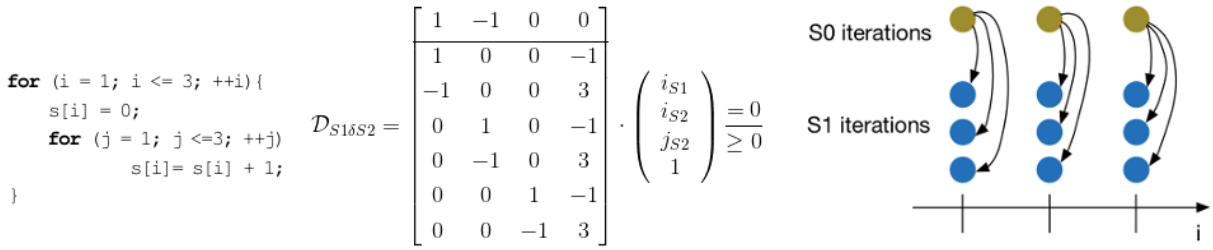


Figure 2.3: An example of a dependence polyhedron. In this example the polyhedron over the iteration vectors (one for the first statement, two for the second) and the scalar part are condensed in a single matrix (notice the 1 after the three iteration vectors). On the right there is a visual representation of the dependencies among the instances of the two statements.

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

Definition 2.1.28. *Polyhedral Model* [113]. The *polyhedral model* of a sequential program consists of a list of statements represented by:

- an identifier;
- a dimension 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.

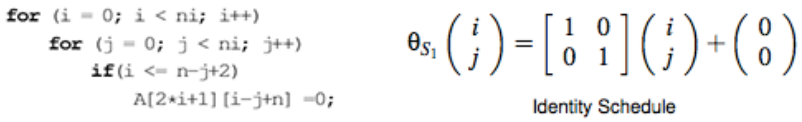


Figure 2.4: A simple schedule example. In this picture, the statement on the left has an *identity schedule*, as the statement instances are trivially the points (i, j) within the statement ID.

Definition 2.1.29. *Affine Schedule [81].* 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 [84].

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 2.1.30. *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 $\forall\langle\vec{x}_R, \vec{x}_S\rangle \in \mathcal{D}_{R,S,f_R f_S.l}$ (*i.e.* for each instance of R and S in dependence, as specified in the corresponding dependence polyhedron $\mathcal{D}_{R,S,f_R f_S.l}$):

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

Definition 2.1.31. *Legal Shedule*. A *schedule* Θ, is *legal* if the *precedence condition* holds.

Lemma 2.1.1. 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 $R \to S$ (definition 2.1.25), a schedule must satisfy [84]:

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

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 $R \to S$ 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 2.1.27.

2.1.2 Polyhedral Transformations

So far, the PM has been described, providing the mathematical toolset which allow 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 [23]:

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 information, 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 statements *instances* which are in dependency. Even though different approaches to this task have been proposed through the years, such as for instance the Omega Test [87], the widely accepted technique it the *Data Flow Analysis* proposed by Feautrier in [47]. 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 2.1.26, nodes are the statements and edges are dependencies between them, and, for each edge, a corresponding dependence polyhedron, described in definition 2.1.27.

The procedure to build the DDG can be characterized by the following algorithm (algorithm 1):

Algorithm 1 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
```

Then, by traversing the obtained DDG, the dependence polyhedra are built, as shown in algorithm 2.

Algorithm 2 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
```

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 the case in which *data reuse* is the major concern, then also Read After Read (RAR) dependencies can be checked [31], 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 so as to impair the correctness, a *legal* schedule must be found, *i.e.* the schedule which optimize the given objective function must be selected within the *legal transformation space* [84, 83]. Hence,

finding a good scheduling algorithm is basically a two-step approach [82]: 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 2.1: Before

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

Listing 2.2: 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 2.3: Before

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

Listing 2.4: 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 loops body, and is also knows as *jamming*. The application of this transformation is safe only if no forward dependency between the two fused loop becomes a backward loop carried dependency. It is used in order to enhanche 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 2.5: Before

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

Listing 2.6: 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 iterations 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 2.7: Before

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

Listing 2.8: After

Loop Peeling This transformation consist in 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 2.9: 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 2.10: After

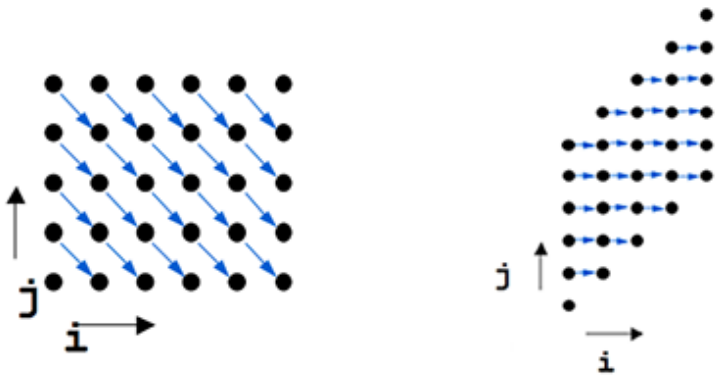
Loop Skewing It takes a nested loop iterating over a multidimensional array, in which each iteration instance of the inner loop depends on previous iterations, and rearranges its array accesses so that the only dependencies are between iterations of the outer loop. Technically speaking, the transformation 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 2.11: Before

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

Listing 2.12: After



(a) Original schedule (b) After skewing

Figure 2.5: Loop Skewing Example. The ID is "skewed" to allow inner loop parallelization.

Tiling Sometimes known as *strip mine and interchange* or *loop blocking*, this transformation is used to enable coarse grain parallelism or 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 [124]. 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 < ni; i++)
    for (j = 0; i < nj; j++)
        A[i][j] = B[j] * C[i];
```

Listing 2.13: Before

```
for (ii = 0; i < ni; i + TILE_SIZE)
    for (j = 0; i < nj; j++)
        for (i = ii; i < ii + TILE_SIZE; i++)
            A[i][j] = B[j] * C[i];
```

Listing 2.14: After

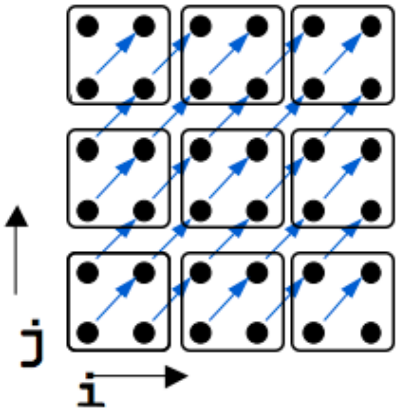


Figure 2.6: Loop Tiling Example. The ID is partitioned into the so called "tiles".

Tiling Hyperplane Method Implemented in the state of art framework known as *PLuTo*, the tiling hyperplane method [31] 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 2.1.2. 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 suggest, 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* [88] 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 [81], 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 [21, 22], which proposed an extended version of the algorithm developed by Quilleré et al. [88]. 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 work have been further improved [111], reaching the ability to scale up to thousand of statements.

2.2 Streaming Systems in FPGAs

Early years FPGAs have been primarily used to implement 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 [104].

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 [114], 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 does 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 enjoys 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.

2.2.1 Streaming Architectures

A stream-based processing system can be viewed as a Multiple Instruction Single Data (MISD) architecture [114], 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.

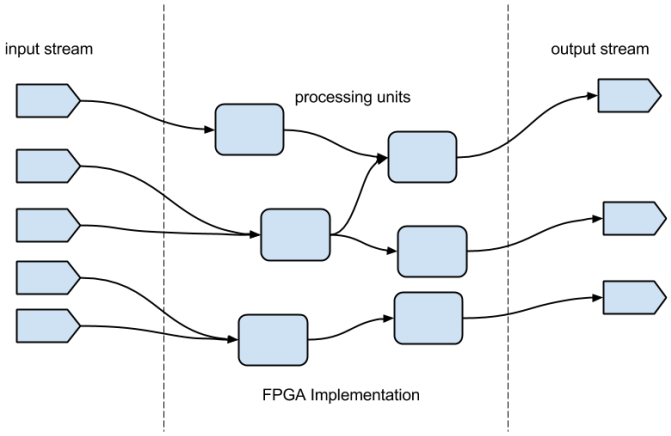


Figure 2.7: Streaming Computing: A General Picture.

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

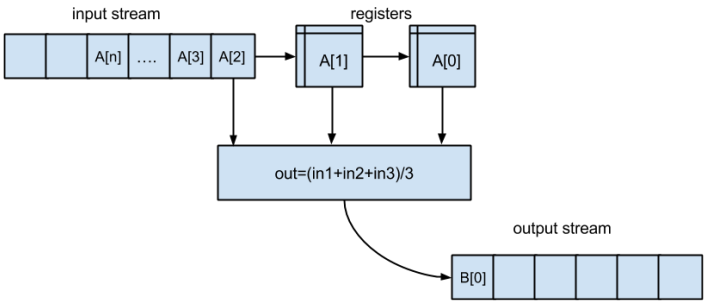


Figure 2.8: Generic Streaming Architecture.

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*, whit computational nodes and memory blocks linked together by *streams*, implemented as dedicated channels (*i.e.* FIFOs), as previously stated.

2.3 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 [36]. 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 [25].

2.3.1 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 [38]. 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 element. 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 algo-

rithm 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.

- **Easy 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 eases 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 processes different RTL design exploitig Domain Space Exploration (DSE).

2.3.2 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 line of code are written, this reduce mistakes and save 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 desing. 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 this 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 application can be accelerated on a reconfigurable computing platforms. The software developers do not write in RTL, so it is required a highly automated synthesis flow from C/C++ to FPGAs.

2.3.3 Evolution

As the design complexity of integrated circuits grows, arises the need of generating circuit implementation from high-level behavioral specifications. The first HLS tools targeted ASICs design, and is CMU-DA [115], 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 languages; 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 be 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 language 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 language are criticized to be only suitable for describing sequential software that run on a Central Processing Unit (CPU). In particular C/C++ language has the following limitations from the hardware point of view:

- does not include constructs to specify accuracy, timing, concurrency, synchronization etc.,
- have 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 [12]), 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, standard C/C++ compiler can compile the code bypassing the pragmas.

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

2.4 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 [62], 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.

2.4.1 Definition

The so called Iterative Stencil Loops are basically a class of iterative algorithms, whose features makes them belonging to the class of SANLP (see section 2.1.1), 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 3.

Algorithm 3 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 efficiently implement them is a great concern, even if not an easy task at all. Indeed, a lot of algorithms for scientific computing, such as [105, 26, 94], as well as image and video processing, such as [33, 51], belong to this class and can be generalized in the form of algorithm 3

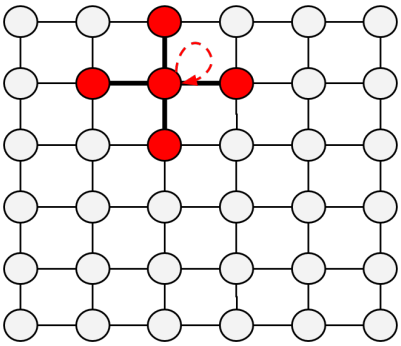


Figure 2.9: An illustration of a generic 5-point 2-Dimensional ISL.

Formal Model

Formally speaking, an Iterative Stencil Loop can be defined as a *5-tuple* (I, S, S_0, s, T) [50] 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 suggest, 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. This weights are stored in separate grids streamed during the computation, which obviously causes an extra memory traffic. This requires special care as stencil 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 2.10a). 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 chosed to approximate large - or even infinite - systems. Obviously, this means that in this case also the boundary change over time, as it is indeed updated during the computation, thus not allowing the optimizations available when dealing with constant boundaries (figure 2.10b).

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

Figure 2.10: ISLs boundary types.

2.4.2 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 arises 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 [41, 62], 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 problem on large grids, not to mention port contention on BRAM [37], 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 [116], lowering performance with repsect to theoretical peak on every device, as the aforementioned, and including also General Purpose Graphic Processing Units (GPGPUs) [98].

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 imply 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 come 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 [105, 94], is an example of such a type of algorithm.

- **Dependent points** When the neighboring elements used in the stencil comes 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 imply that no - or at least non

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 [94] is a perfect example of this category. A parallel version of this method has been however developed, namely *red-black* Gauss-Seidel [68], 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 [116].

2.4.3 State of the Art

The implementation challenges discussed so far in section 2.4.2, 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 2.1.2), 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 [90], which are shown below.

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 [62]. 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 [52], also leveraging specific Application Programming Interfaces (APIs) such as OpenMP [40].

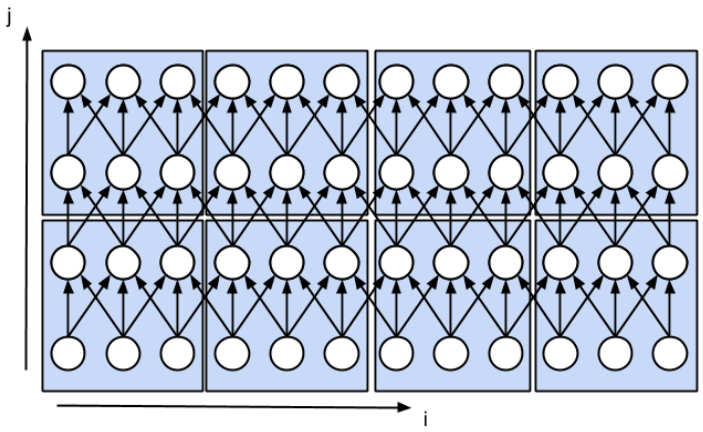


Figure 2.11: Single Iteration Tiling.

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* [74, 66, 60], 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 [116].

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 increase the overall data reuse. However, in order to make

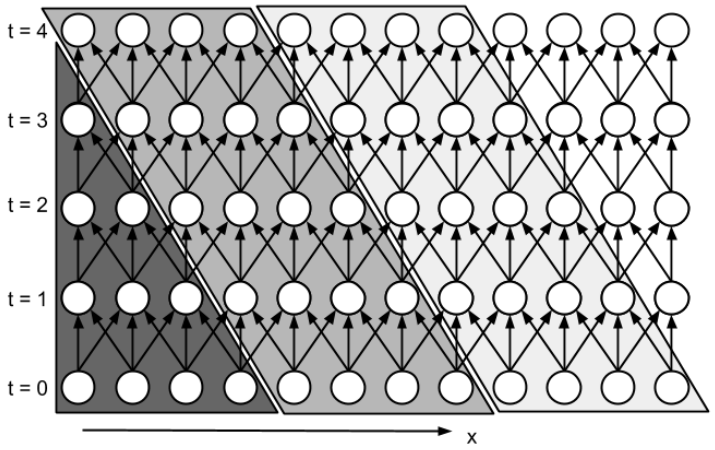


Figure 2.12: Time Skewing.

tiling legal, loop skewing (see section 2.1.2) 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 [91], 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 [90], as well as on the form of the tile [106, 91], which can be a major concern especially on FPGAs [126]. Whit respect to the previously mentioned strategy, time skewing can provide better cache hit rates and effectively reduce processor idle time caused by the ISLs memory boundedness [121].

As for the previous tiling strategy, even in this case blocks distribution among different PEs is possible [14], 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 [75], 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 [34], in which a domain-specific compiler is proposed, namely *Caracal*, able to perform unrolling of the time loop and fuse accordingly the stencils, with the same effects as of [75].

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 [117, 98, 110]. 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.

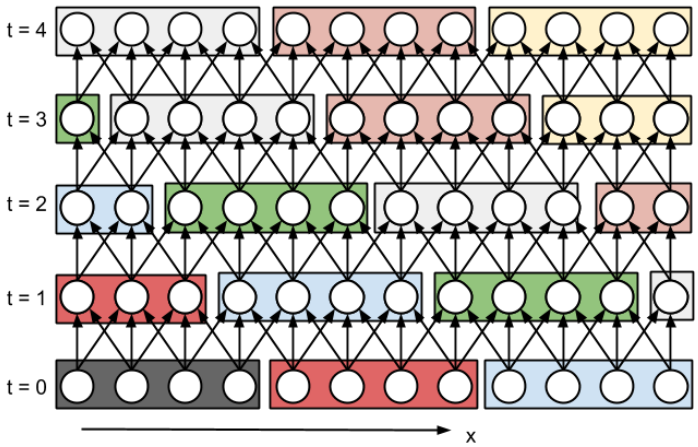


Figure 2.13: Wavefront Parallelization.

Although in [90] 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 [95, 96, 79], and by far this approach is the most promising one with respect to the previous two, as it has been proven to be scalable [95, 96, 79], and comes with no communication overhead.

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

DSLs Based Optimizations

Another important approach towards optimization, extensively used in literature, and which is becoming increasingly popular, consist of the exploitation of Domain Specific Languages (DSLs) and ad-hoc frameworks. As indeed 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, at a cost of losing broad applicability (although in some cases it could be still technically possible), the ability to define a problem within the specific application domain in such a way that some features are explicitly expressed, enabling a whole sort of transformations, manipulations and optimizations simply not possible - or hard to achieve - with GPLs.

Currently, the ISLs domain can count a number of available DSLs, each one with its own peculiarities. For instance, PATUS [35] is able to achieve high performances by auto-tuning, targeting different hardware architecture, while Pochoir [109] provides a C++ template library based on a divide-and-conquer skeleton which is then translated into Cilk [30], a C/C++ extension designed for multithreaded parallel computing. ExaStencils [69] 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 [107] abstract from Scala with the aim of making stencil programming easier, and use 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 [125], 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 [60], 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 [118] a single high-order function specified in Haskell, and specifically in CλaSH [17], 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 transformation 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 [92] 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) [71], 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 [37] 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 [65] 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 proves as an FPGA custom architecture can deliver power efficiency much higher than traditional computing devices.

In [100] 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 prefetch 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.

3

A Scalable Hardware Accelerator for Iterative Stencil Loops

In this Chapter the thesis proposal for a custom hardware accelerator for Iterative Stencil Loops (ISLs) is presented, from a clear statement of the problem, to the proposed solution. In particular, Section 3.1 provides an in-depth analysis of the challenges and the issues related to the implementation of ISLs, while Section 3.2 presents the problem statement along with the thesis contribution. Finally, Section 3.3 introduces the proposed solution from a high level perspective, supplying the fundamental principles of the work, in Subsection 3.3.1; a description of the hardware accelerator, in Subsection 3.3.2; a definition of the accepted input code, in Subsection 3.3.3; and a comparison with existing works, in Subsection 3.3.4.

3.1 The Issue of Finding an Efficient Implementation

Numerical methods for Partial Differential Equations (PDEs) solving employed in weather and ocean modeling [72, 29], fluid dynamics [44, 97], quantum dynamics simulations [77, 80], heat diffusion [53], geometric modeling [64] and non-equilibrium statistical mechanics [32], but also seismic simulations [67, 93] and cellular automata [78], as well as multimedia/image-processing applications such as [51, 57], gaussian smoothing [102] and Sobel edge detection [16], repre-

sent only a fraction of a wide number of applications that share the same computational nature, in which a series of sweeps (time-steps) are performed over a regular grid, where points are updated using a fixed nearest neighbor pattern. This kind of kernel is known as ISL, as already discussed in the previous chapter, in section 2.4. Since stencil computations are characterized by this regular computational structure, they are the perfect candidate for automatic compile-time analysis and transformation aimed at improving their performance. However, there are some important remarks that have to be done in order to understand why, even considering this possibility, an optimal solution is yet to be found.

As already discussed in section 2.4.2, the memory boundedness is *the* problem of ISLs, and it is transversal to all the computing architectures, although in slightly different manners. As a matter of fact, even if the problem has been thoroughly investigated, and many techniques for alleviating it have been proposed, all the proposed approaches are either limited in their applicability, or unsatisfactory in terms of the performance gains compared to the theoretical peak the available computing architectures can offer.

All the state of the art techniques, presented in section 2.4.3, tackles this issue trying to balance the computation and the memory transfers, both with in-core and inter-core optimizations.

In the first case, the obvious approach is to enhance data locality, which can lead to a reduced number of memory accesses. This can be achieved exploiting either the locality within a single time-step (spatial locality) or across multiple time-steps (temporal locality). Although it may seem that this could be an effective solution, both the approaches are actually hard to implement, as they require a proper tuning to match the target architecture characteristics, scratch-pad memory size as first thing. The difficulty escalates considering also that in the general case algorithm developers have profound knowledge of the application domain, but they often lack a proper understanding of the underlying architectures. Some solutions in this sense have been actually proposed, namely *cache oblivious* algorithms [86], but usually their optimality is only theoretical - asymptotical - and even if in principle they promise to be “architecture-inpependent”, they instead

require tuning of parameters in real cases.

The situation is even more complicated when trying to overcome the memory boundedness with parallelization-oriented solutions. For instance, within the spatial domain of a single time-step, many ISLs are highly - trivially, indeed - parallelizable, simply because the update of a grid point is totally independent from the others, enabling to run these updates concurrently on many processing units. However, this comes with an important drawback, as the increase in the number of those computational units corresponds to a dramatic increase of the memory bandwidth demand, due to the necessity to both concurrently feed the computational units with the needed data and synchronize their outputs, effectively bounding the achievable performance by the available memory bandwidth instead of the computational capacity.

In the context of multi-core architectures, the computation can be made less dependent on memory bandwidth performing both parallelization and locality enhancement together. However, this often lead to *pipelined startup* [18], *i.e.* not all processors are busy during parallelized execution, resulting in bad resource usage and consequently low power efficiency. Also, the lack of full concurrency at the start can impact negatively the asymptotic degree of parallelism when tyring to scale up both data set size and number of computational units [122].

In theory, a solution for the memory issue could then be to employ architectures equipped with high bandwidth memories, such as General Purpose Graphic Processing Units (GPGPUs). In practice, even GPGPUs suffers of memory boundedness. In fact, the data from the off-chip memory is transferred in contiguous blocks, and therefore high bandwidth can be achieved only when read requests by concurrent threads in a warp fall within such contiguous blocks. When optimizations aimed at distributing the ISL computation, such as tiling, are employed, the result is usually that threads perform sparse memory accesses, but also that they follow different control paths, causing *branch divergence*, which is another source of inefficiency. Last but not least, in GPGPUs scratch-pad memory is usually implemented as a banked memory system. If concurrent threads request data from the same bank, the conflict will result in the serialization of the

read requests. In all these scenarios both performance and power efficiency are impaired, and do not achieve their respective maximum.

Another important problem that arise when inspecting the state of the art of ISLs implementation is undoubtedly the limited applicability. Indeed, the great part of the currently available approaches often address only a fraction of the entire ISL domain, targeting for instance only 1-D [59, 39], 2-D [99, 65, 59] or 3-D [125, 100, 92] grids. Also, the vast majority is simply not able to deal with ISLs that have spatial dependencies among points within a single time-step (see section 2.4.2 for an explanation of what they are), which, however, constitute an important part of this class of algorithms.

As previously stated, the scalability of ISL algorithms in large-scale clusters is limited by data dependency between the distributed workloads. This is mainly due to architectural limitations, which by the way also narrows the optimizations exploration space, pushing them all in the same direction. For this reason, architecture customization can be considered a valid method towards the optimal ISLs implementation. In particular, the acceleration of stencil applications using Field Programmable Gate Arrays (FPGAs), can lead to exploit not only fine-grained parallelism, as well as limit the memory boundedness issue, but can also enable scaling over multiple FPGAs nodes. The state of the art regarding custom FPGAs architecture has been already explored in section 2.4.3, there are however some further considerations to make, due to the inherent complexity of these solutions. First of all, a lot of architectures are tailored to specific algorithms [33, 55], which means that they are in general not applicable to the entire class of ISLs. Also, building an hardware architecture is in general a hard task, usually done by hand (or lightly assisted), and because of that, error prone. Furthermore, this process can take a non negligible amount of time, and despite the efforts made, result in improvements so small that it is simply not worth the hassle. This means that a widely applicable, efficient, but nevertheless automatic solution must be proposed, in order to take advantage of hardware design in the most effective way.

3.2 Thesis Contribution

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 does 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 ISLs implementation. The aim of the work proposed in this thesis is instead to address all the presented issue 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 FPGAs. 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 enable the execution of multiple time-steps in one pass ensure 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 Polyhedral Model (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 provided:

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 Streaming Stencil Time-step (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.

3.3 The Proposed Solution

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 hardware accelerator in all of its components, a set of constraints for the input code of the proposed solution, and a comparison with existing works.

3.3.1 Fundamental Principles

Let us first expose the fundamental principles on which the proposed work rests its foundation, since introducing them clearly is a crucial precondition to understand the work completely.

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.

Scalability

In assessing the quality of an implementative solution, 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 instead completely focused on scalability (practically measured in chapter 5), 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 queue that constructs a deep pipeline. Because the depth of the queue does not influence the bandwidth requirements, the computing performance can be increased with a constant memory bandwidth by connecting more SSTs for a longer queue.
- 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 2.4.2), 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. We do not account directly for parallelism, 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 section 3.1, 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.

3.3.2 A General Overview of the Proposed Hardware Accelerator

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 embody exactly 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 hardware accelerator can be viewed, at every level of granularity, as a composition of *independent* modules, that communicate over First In First Out (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 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 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 section 4.4.1. In this case the *mux* is responsible to break this loop when the total number of time-steps has been executed.

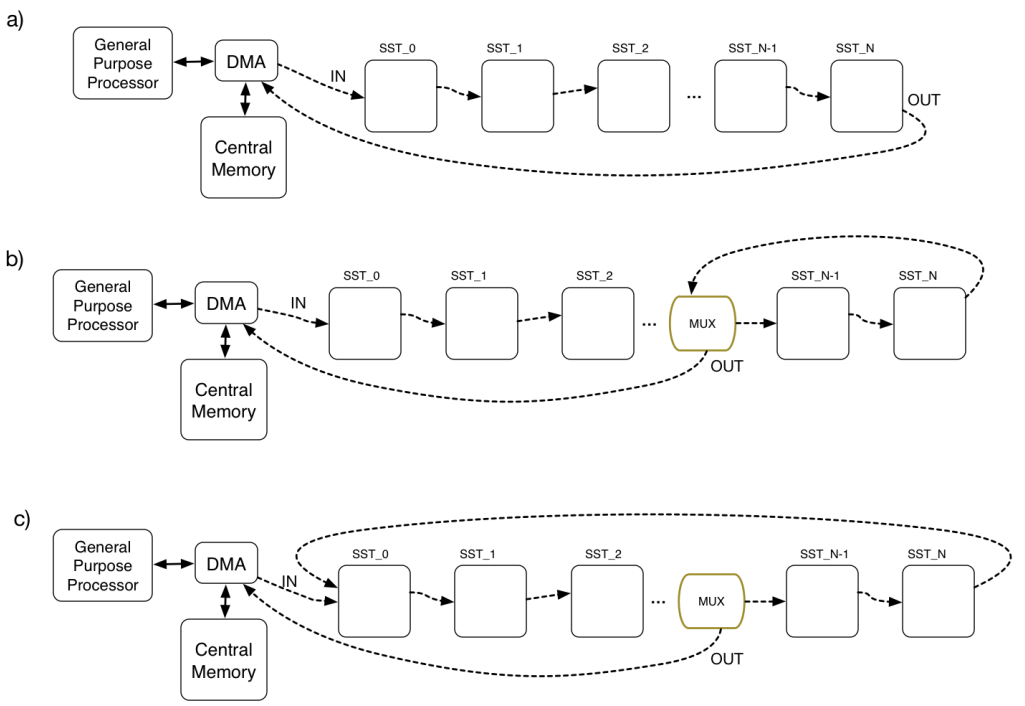


Figure 3.1: The high level scheme of the proposed hardware accelerator. The three different versions represent the three distinct described cases: the first (a) is the standard case, the second (b) is the case in which the queue length is not an exact divisor of the total number of ISL time-steps, the 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*.

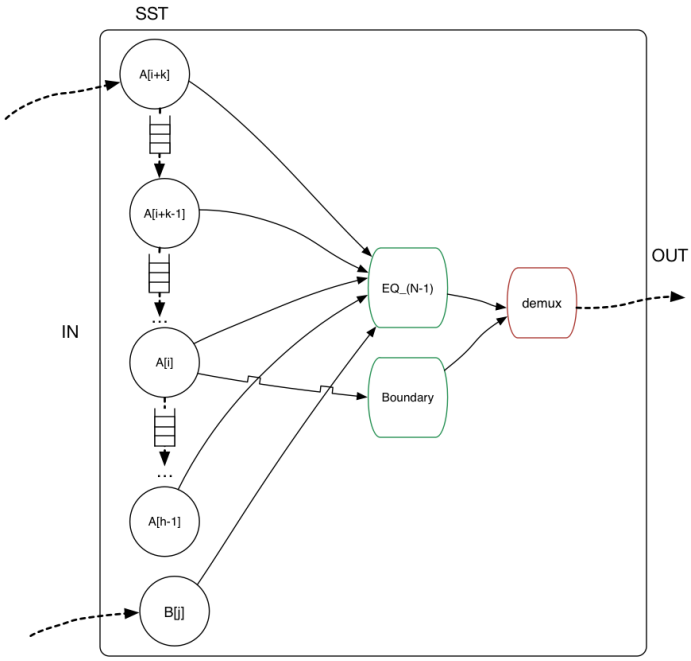


Figure 3.2: A general scheme of an SST.

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 represents 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.

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, the one actually responsible for the production of the SST output, will actually always decomposed into two parts, one demanded to compute the ISL output, which we may refer to as *computation part*, and one which transfer the boundary from the memory system. 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.

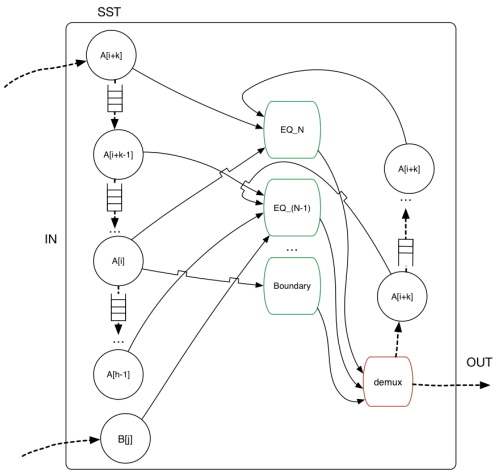


Figure 3.3: An SST for ISLs with spatial dependencies.

We claimed previously 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 needed data to all the SSTs, additional streams must be added, in order to transfer the input data within the queue. In this case then, the *filters chains* referred 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.

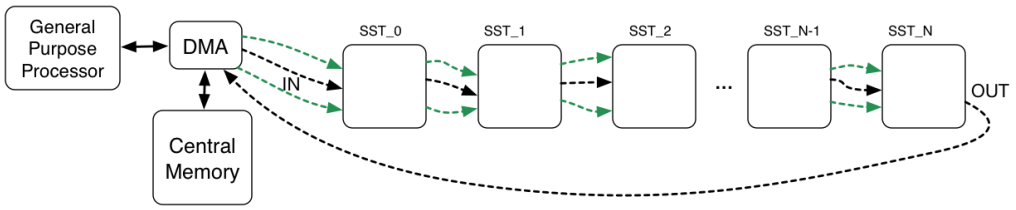


Figure 3.4: An example of the accelerator for an ISL with multiple inputs. The green arrows represents the additional streams. As described in the text, the last SST has only the actual output stream.

3.3.3 Some Considerations on the Input Code

We already stated in section 3.3.1 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 Static Affine Nested Loop Programs (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 assignment 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 does 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 chapter. 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.

3.3.4 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.

PM-based C-toFPGA flow

Although High Level Synthesis (HLS) have seen an intense evolution, as already described in section 2.3, 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 [85, 70] and [126], which employ the PM as optimizing

engine. The aim of all this 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 [85] use tiling hyperplane transformations to expose data locality as much as possible, and then carefully manage on-chip buffer to enable data reuse and pre-fetching. The generated 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 [70] PolyOpt/HLS has been extended with optimizations tailored to solve the resource contention on memory banks 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 [126] the authors extends the state of art framework *PoCC* [7] (a framework for polyhedral optimizations that 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 communications 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 [8], 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.

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 [37], 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 [37], even if technically they are implemented differently. There are however two observations that have to be made, since the work in [37] 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 chapter 5.
- 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 does not take into account practical constraints such as the available bandwidth.

The second comparison can be made with Maxeler [6]. 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 Central Processing Units (CPUs) and DFEs, and DFEs configurations are created using Maxeler's MaxCompiler. To create applications exploiting DFE configurations, an application must be explicitly splitted into three parts:

- Kernel, which implement 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 3.3.3, automatically.

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 [75], where the target is hardware design, and [34], where the target is canonical CPU-based architectures. In [75], 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 [34] 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 [95], 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 [54], 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.

4

Proposed Design Flow

The purpose of this Chapter is to provide a thorough description of the proposed methodology. In Section 4.1 a general overview of the proposed *design automation flow* is presented, while Section 4.2 explains briefly the *pre-processing* phase, needed to reshape the input code in order to be manipulated thereafter. Section 4.3 details the first part of the process of automatic production of the Iterative Stencil Loop (ISL) accelerator, namely *Streaming Stencil Time-step (SST) Microarchitecture Derivation*, along with some important remarks on the SST itself. Then, Section 4.4 provides a description of the second part, that is the *SSTs queuing* technique.

4.1 Design Automation Flow

As already described in section 3.3, the proposed methodology is a 2-step process. The first step consists of deriving the microarchitecture that is demanded to implement a single iteration - *i.e.* a *time-step* - which we called Streaming Stencil Time-step (SST). This can be viewed as the *basic building block* of the system. The second part addresses the system construction, in which SSTs are arranged in a *queue* fashion, unleashing the power of the methodology by enhancing all the peculiarities of a single SST, with an eye also on scalability.

The 2-step methodology has been employed to develop a *design automation flow*, that takes as input the ISL's SCoP, 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. This is done in order to simplify the architecture derivation, as already stated in section 3.3.3. An overview of the proposed flow can be seen in figure 4.1.

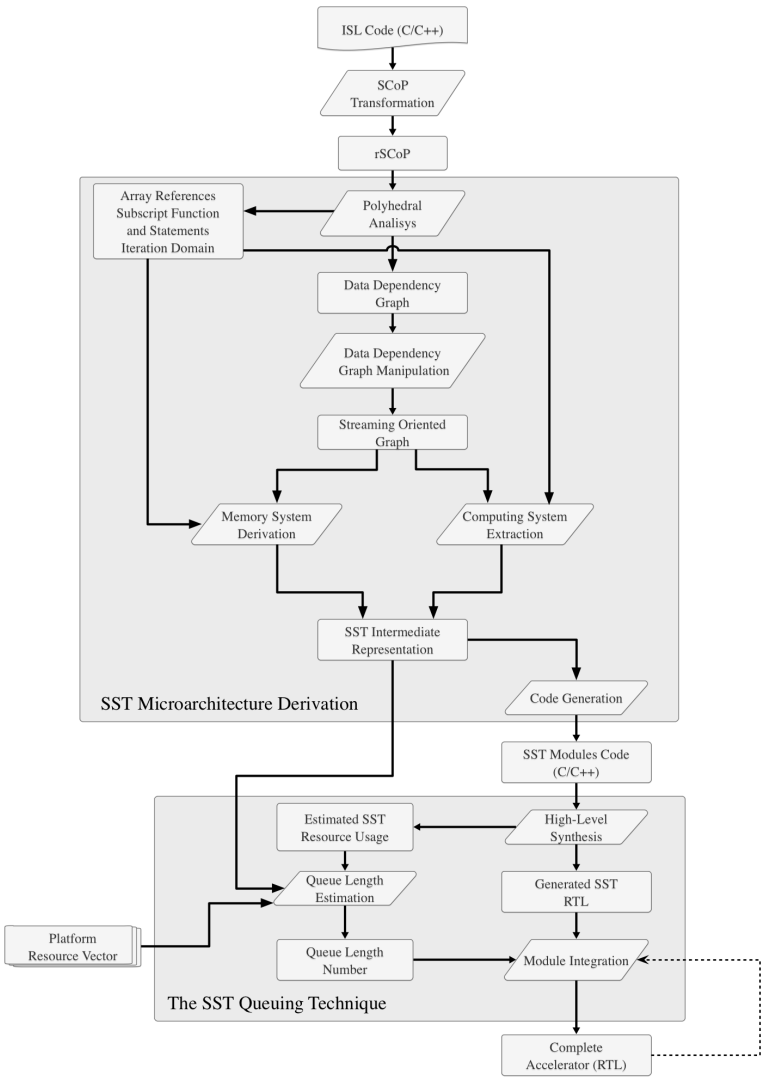


Figure 4.1: The Proposed Design Automation Flow.

Let us briefly describe the two macroblocks of figure 4.1. 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 Intermediate Representation (IR) of the input code, and also the cor-

responding Data Dependency Graph (DDG), which is crucial for the entire SST derivation process.

- The second part consist of an ad hoc manipulation of the obtained DDG in order to obtain the skeleton of the SST, which we called *streaming-oriented graph*.

- After that, two concurrent phases take place. The first is the *memory system derivation*, that employs the polyhedral IR along with the streaming-oriented graph in order to derive the *chains* described in section 4.3.3 and the corresponding involved streams. The second is the *computing system extraction*, whose function is to shape the computing system that effectively realize the Iterative Stencil Loop (ISL) computation.

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 High Level Synthesis (HLS). 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 Register-Transfer Level (RTL) of the resulting hardware accelerator.

Although not actually realized, we will prove in the following sections that this design flow is completely automatable, by proving that each component part is itself automatable.

4.2 Pre-processing Phase

In order to allow the subsequent manipulations, we claimed that a *pre-processing phase* is essential, since it could be that the original Iterative Stencil Loop (ISL) code is not in a suitable form. Here we describe how this can be easily accomplished by employing the Polyhedral Model (PM) and the state of the art tools, without particular modifications.

Although it is a situation that is very unlikely to be found in canonical ISLs codes, it can happen that the ISL's SCoP contains conditionals which are *affine functions* of the outermost loop, that is indeed the time dimension (see section 2.4.1). Notice that the reason why affine conditionals on the inner loops are simply not allowed at all are already discussed in section 3.3.3.
When such a situation is in place, the code *must* be transformed, otherwise the microarchitecture derivation steps may lead to unwanted behavior. The reason why this pre-processing is needed is pretty straightforward: 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. When deriving a streaming microarchitecture which is demanded to implement the code of a *single stencil iteration*, this situation is unacceptable. However, assuming that n is the number of the mutually exclusive code parts, we could in principle derive n different microarchitectures, and leave their actual usage to the *host processor*, which employ one microarchitecture or the other according to the given time-step.
To deal with this case without deeply modifying the already complex flow, a simple solution is to apply the *index-set splitting* transformation along the *only* time dimension (the outermost loop). The affine conditionals can be used to effectively drive the splitting on the original loop nest, in order to obtain a different loop nest for each mutually exclusive part. If such a transformation is performed, it can be safely assumed that the conditionals can be removed from the obtained code.

```
for (t = 0; t < T; t++){
    for (i = 1; i < N-1; i++){
        if(t < T/2)
            A[i] = (A[i+2]+A[i+1]+A[i]+A[i-1]+A[i-2])/5;
        else
            A[i] = (A[i+1]+A[i]+A[i-1])/3;
    }
}
```

Listing 4.1: The original ISL code

```
for (t = 0; t < T/2; t++)
    for (i = 1; i < N-1; i++)
        A[i] = (A[i+2]+A[i+1]+A[i]+A[i-1]+A[i-2])/5;
for (t = T/2; t < T; t++)
    for (i = 1; i < N-1; i++)
        A[i] = (A[i+1]+A[i]+A[i-1])/3;
```

Listing 4.2: After index-set splitting, two rSCoP are obtained

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 Reduced Static Control Part (rSCoP), since they actually still belong to a single SCoP, but despite this we treat each of them individually. Such an rSCoP is the input of the following block within the design automation flow, namely the *SST Microarchitecture Derivation*. Notice that from now on, we will refer to both ISL in general and rSCoP indistinctly, since for our purposes they can be thought as equivalent.

4.3 The SST Microarchitecture Derivation

As shown in figure 4.1, the first part of the proposed methodology is the derivation of a Streaming Stencil Time-step (SST) microarchitecture from the input rSCoP, which could pass the pre-processing phase completely unchanged (thus being the entire Static Control Part (SCoP), instead of a fraction of it). To work properly, this macroblock relies heavily on the PM, that allow all the analysis and manipulations which will be explained in a moment. For this reason, we will assume that the reader has at least a basic knowledge of what the PM is, and how it can be employed to perform static analysis on Static Affine Nested Loop Programs (SANLPs), and by extent on ISLs. The reader may refer to section 2.1

for a comprehensive overview of the PM, which provides all the concepts and definitions used in this section.

In order to support the following steps with concrete examples, we will employ two sample ISLs, shown in the listings below.

```
for (t = 0; t < T_Step; t++)
    for (i = 1; i < N-1; i++)
S0:     A[i] = 0.2*(A[i-1]+A[i]+A[i+1]);
```

Listing 4.3: Sample ISL number 1

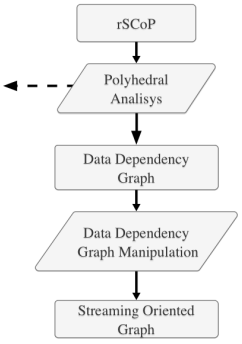
```
for (t = 0; t < T_Step; t++){
    for (i = 1; i < N-1; i++)
S0:     B[i] = C[i]*(A[i-1]+A[i]+A[i+1]);
    for(i = 1; i< N-1; i++)
S1:     A[i] = B[i];
}
```

Listing 4.4: Sample ISL number 2

There are essentially three reasons for which both ISLs are deemed useful:

- They are representative - respectively - of the two basic types of ISLs, since the one in listing 4.3 has *spatial dependencies* between grid points, while the other in listing 4.4 has only dependencies along the time dimension (see section 2.4.2);
- They cover both cases in which: *a)* the stencil array is the only input (listing 4.3), or *b)* there are more than one input arrays (listing 4.4)
- They are really simple, being both only 3-point stencils and also monodimensional. We decided to use those samples since we considered a complex example simply not suitable to support effectively the description.

From now on, we will refer to the sample in listing 4.3 as *sample1*, while the sample in listing 4.4 will be referred as *sample2*.

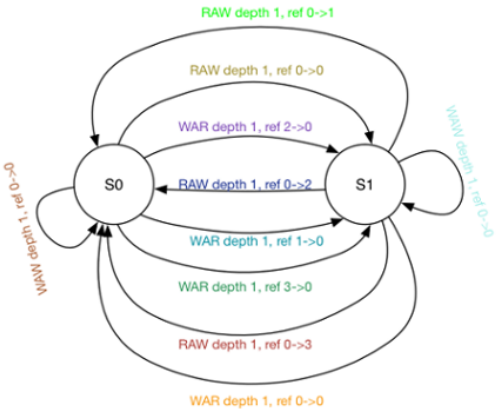


4.3.1 Streaming-oriented Graph Construction

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 microarchitecture. This graph, as it will be shown in a moment, is nothing more that the result of a manipulation of the Data Dependency Graph (DDG).

The first task that must be performed is the *data dependency analysis*, in order to

produce, for a given rSCoP, the corresponding DDG. This can be easily achieved with the already available tools in the state of the art. The result of this process for *sample2* is shown in figure 4.2.



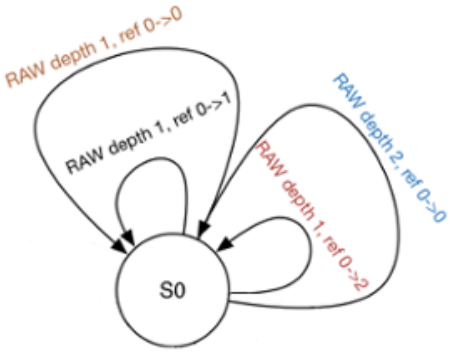

```
S0:   B[i] = C[i] * (A[i-1] + A[i] + A[i+1]);
ref:   0       0        1        2        3
op:    W       R        R        R        R
S1:   A[i] = B[i];
ref:   0       0
op:    W       R
```

```
digraph G {
# Data Dependence Graph
# Generated by Candl 0.6.3 64 bits
  S0 -> S0 [label=" WAW depth 1, ref 0->0 "];
  S0 -> S1 [label=" RAW depth 1, ref 0->0 "];
  S0 -> S1 [label=" WAR depth 1, ref 1->0 "];
  S0 -> S1 [label=" WAR depth 1, ref 2->0 "];
  S0 -> S1 [label=" WAR depth 1, ref 3->0 "];
  S1 -> S0 [label=" RAW depth 1, ref 0->1 "];
  S1 -> S0 [label=" RAW depth 1, ref 0->2 "];
  S1 -> S0 [label=" RAW depth 1, ref 0->3 "];
  S1 -> S0 [label=" WAR depth 1, ref 0->0 "];
  S1 -> S1 [label=" WAW depth 1, ref 0->0 "];
# Number of edges = 10
}
```

Figure 4.2: An example of a complete DDG. The graph is computed for *sample2* in listing 4.4. On the right, there is the output of the state of the art tool for *polyhedral dependency analisys Candl* [1], while on the left, there is the graph in its graphic form.

However, the DDG, as it is, contains unnecessary information. Since in fact our purpose is to obtain a *streaming-oriented graph*, the only dependencies that must be taken into account are the Read After Write (RAW) dependencies (*true data dependency*), *i.e.* the one enforced by the data flow. Indeed:

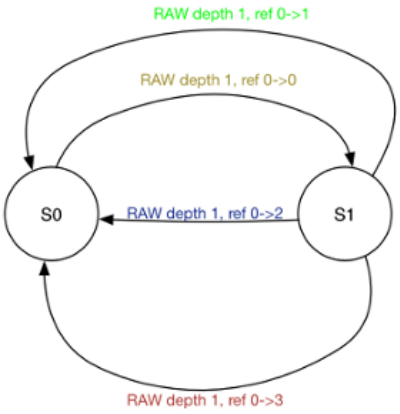
- As we do not operate any alteration of the control flow, *i.e.* the execution follows the original *schedule*, an SST can be viewed as an *in-order* microarchitecture [58], which means that we can safely neglect the Write After Read (WAR) dependencies (*antidependency*);
- Even though the observations made above are not sufficient to neglect the Write After Write (WAW) dependencies (*output dependency*), just by adding to the previous considerations the fact that the data flow is *completely under control* - as it is determined at compile time - we can discard also them in the analysis process.




```
S0:   A[i] = 0.2 * (A[i-1] + A[i] + A[i+1]);
ref:   0            0       1       2
op:    W            R       R       R
```

```
digraph G {
# Data Dependence Graph
# Generated by Candl 0.6.3 64 bits
  S0 -> S0 [label=" RAW depth 1, ref 0->0 "];
  S0 -> S0 [label=" RAW depth 2, ref 0->0 "];
  S0 -> S0 [label=" RAW depth 1, ref 0->1 "];
  S0 -> S0 [label=" RAW depth 1, ref 0->2 "];
# Number of edges = 9
}
```

(a) The proper DDG for *sample1*, listing 4.3.




```
S0:   B[i] = C[i] * (A[i-1] + A[i] + A[i+1]);
ref:   0       0        1       2       3
op:    W       R        R       R       R
S1:   A[i] = B[i];
ref:   0       0
op:    W       R
```

```
digraph G {
# Data Dependence Graph
# Generated by Candl 0.6.3 64 bits
  S0 -> S1 [label=" RAW depth 1, ref 0->0 "];
  S1 -> S0 [label=" RAW depth 1, ref 0->1 "];
  S1 -> S0 [label=" RAW depth 1, ref 0->2 "];
  S1 -> S0 [label=" RAW depth 1, ref 0->3 "];
# Number of edges = 10
}
```

(b) The proper DDG for *sample2*, listing 4.4.

Figure 4.3: The DDG with the only RAW dependencies, the only dependencies we take into account. As for image 4.2, on the right there is the output of *Candl*, while on the left the DDG is its graphic form.

Figure 4.3 shows the proper DDG (the one we will employ) of the two samples, without WAW and WAR dependencies.

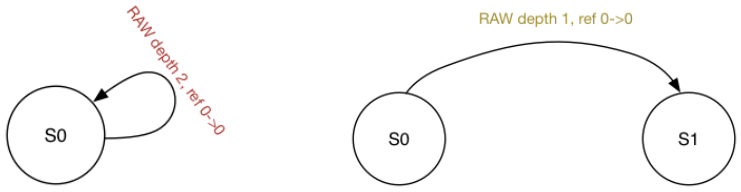
After the DDG extraction, the *graph manipulation* phase takes place.
First of all, the DDG must be pruned further, because it could still contain dependencies that are not significant in the SST microarchitecture derivation. In particular, the dependencies carried by the time dimension (the one marked with *depth 1* in the figure 4.3) *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 marked with depth 1 (along the time dimension) can remain after this pruning task. This situation can occur whenever the rSCoP loop nest is *imperfect*, such as in the case of *sample2*, as it can be that the flow dependencies within the

same time-step are carried exactly by the time dimension. Those dependencies are in fact enforced by the execution order of the statements, and *not* carried between a time iteration and the subsequent, which is why they are significant for our purposes.
As a result, we can define the following conditions for an edge to be removed during the pruning task:

Definition 4.3.1. *DDG Edge Removal Conditions*. Let us consider a DDG $G = (n, e)$, 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 time dimension (depth 1)*, and:

- e is a self-loop or
- e is directed from n_i to n_j and $i > j$.

The result of this process is shown for both samples in figure 4.4.



(a) *sample1*, listing 4.3. (b) *sample2*, listing 4.4.

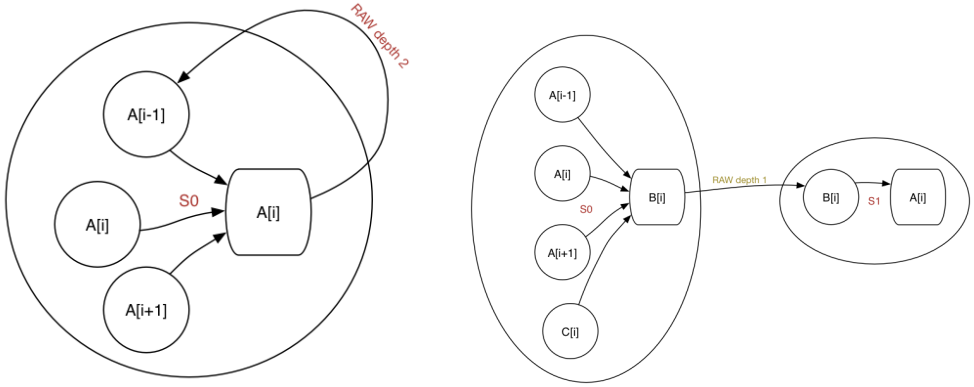
Figure 4.4: The DDG after the pruning process, whose conditions are given in definition 4.3.1

The next step is the key part of the *DDG manipulation* phase, as it will effectively turn the original DDG into the so called *streaming-oriented graph*. Firstly, *each* node (which is indeed a statement) of the DDG is *expanded* in the following way:

1. Each array reference - read or write for now does not matter - becomes a new node. Since for each array assignment - *i.e.* statement - the polyhedral analysis (indeed the parsing that takes place at the beginning) is able

to identify the read and write operations, we employ this information to connect those 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 represents the statement execution, hence, they are associated with the statement's assignment (*i.e.* formula) and its Iteration Domain (ID).

2. The original edges of the DDG are now connected, rather than with the entire statement, to the specific node - *i.e.* array reference - involved in the corresponding dependence. The result of this two operations is displayed for both *sample1* and *sample2* in figure 4.5.



(a) *sample1*, listing 4.3. (b) *sample2*, listing 4.4.

Figure 4.5: The expanded version of the DDG for both samples. In order to easily distinguish them, *read* and *write* nodes are represented with different shapes.

The last step towards the construction of a *streaming-oriented graph* consists of 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:

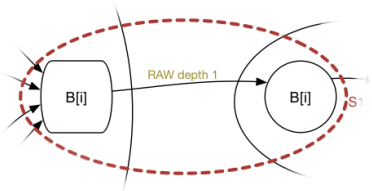


Figure 4.6: The dependence within the red circle is an example of the so called "copy" dependency.

- 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 ID on the reference *subscript function* - matches the ID of W;
- if N is a *write* node, its ID 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 following pseudocode:

Algorithm 4 Iterative Reduction of the *Streaming-oriented* graph

Input: the streaming-oriented graph $G = (n, e)$
Output: the reduced version of G

```
R = 0
for all write node n ∈ G do
    if n has only one outgoing edge e∧e is directed to a node n′ in a "copy" dependence relation
then
        R ← R ∪ (n, e, n′)
    end if
end for
while R ≠ 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∧e is directed to a node w′ in a "copy" dependence relation
then
        R ← R ∪ (w, e, w′)
    end if
end while
```

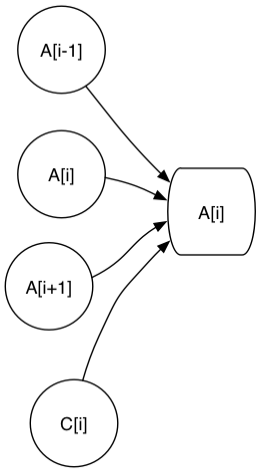
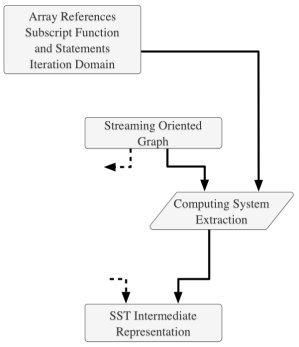


Figure 4.7: The *Streaming-oriented Graph* of *sample2*, listing 4.4.

The obtained reduced graph no longer contains the notion of statement, and is completely different from the orignal DDG. This is the skeleton of the SST microarchitecture, and we called it *Streaming-oridented Graph*. However, the *streaming-oriented* graph, as it is, represents only a part of the whole picture. In order to have the Intermediate Representation (IR) of an SST, both the *computing system* and the *memory system* must be properly characterized. In the following two subsection we will explain the procedures to achieve this goal.

4.3.2 Computing System Extraction



The *streaming-oriented* graph has provided the only write nodes that will effectively become computation modules of the *computing system*. We already stated that to work properly, within an SST - and between SSTs - the streams entering the *filter chains* of the *memory system* must be in the form of the entire array. Given that premise, it is obvious that, as already described in section 3.3.2, the boundaries must be explicitly managed. To do so, the following steps have to be executed:

1. Firstly, we make important remark, that comes directly from both the nature of an ISL computation and also the consideration made in 3.3.3: the array updated within the statement with the highest index - *i.e.* the last in the syntactic order - *must* be the stencil array. We employ this information to identify explicitly 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. Note that how the boundary is managed will be explained when the memory system will be derived;

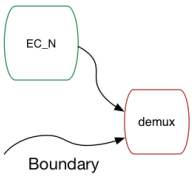


Figure 4.8: A visual representation of the instantiation of a *demux*.

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 portion of the array which will be updated with different formulae. We call these portions *equivalence classes*. Since the write nodes themselves are indeed responsible of the update of these regions of the array space, we will refer indistinctly to both regions and corresponding write nodes as *equivalence classes*. We now precisely define what an equivalence class is, as it will be also needed later.

Definition 4.3.2. *Equivalence Class*. An equivalence class 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*;
- Each point of P 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 DDG previously described, no cycles between statements can be in place. The only kind of cyclic dependencies that can be admitted 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. This indicates the presence of spatial dependencies between grid points. In this case, the write nodes involved in this cyclic dependencies require a further treatment. From now on we will refer to:

- *write* nodes that enjoy this characteristic as *cyclic-write* nodes,
- *read* nodes which concur in the cyclic dependency as *cyclic-read* nodes.

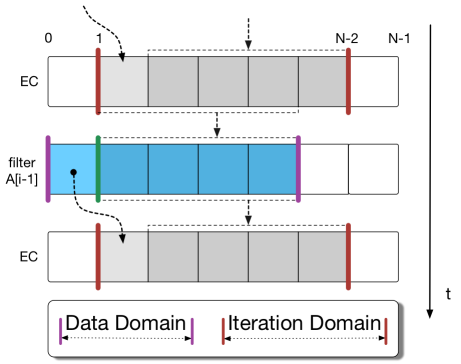


Figure 4.9: An illustration of the cyclic dependencies between the output of the *cyclic-write* node and the *cyclic-read* node (filter) A[i-1] of *sample1*, listing 4.3.

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 ID. Hence, every cyclic dependence is indeed realized as an edge from the *cyclic-write* node entering a *cyclic-read* node whose array reference *data space* (also called *data domain*), which is the *image* of the *cyclic-write* node ID on the reference *subscript function*, partially overlaps with

the boundary. This means that for the given *cyclic-read* node, there is a part of the data space which will come from the boundary, and a part which instead is given by the *cyclic-write* node's output, which in turn means that it will take data from two different input streams, one of which being the *cyclic-write* node's output itself. Therefore, *cyclic-read* nodes will be implemented as two distinct filters, each belonging to a different chain, leading the *cyclic-write* node to have different portions of its ID dependent on different sets of filters.
The actual arrangement of the memory system will be discussed in the next subsection, for now we simply mark the *cyclic-read* nodes with two symbolic values, one representing the part of the data domain which overlaps with the boundary, one representing the part which overlaps with the *cyclic-write* node ID, since they will be needed in a moment. Here, we instead focus on the implications of such a condition on the computing system. What happens is that the ID of each *cyclic-write* node is partitioned into subsets which are indeed dependent on a different set of filters, even if the formula is actually the same. Hence, they are indeed different *equivalence classes*, as previously defined, even though we will refer to them in the rest of this subsection as *sd-equivalence classes* (the prepended "sd" stands for *spatial dependence*), to differentiate them from the one previously derived. To partition the original *cyclic-write* node ID, we need three basic information:

- the number of sd-equivalence classes,
- the ID of those sd-equivalence classes, each of them being a partition of the original ID
- the set of input filters of each sd-equivalence class

The number of sd-equivalence classes can be computed with a simple formula. We argue this claim with the following observations:

1. an sd-equivalence class can exist *if and only if* there is a spatial dependence between computed points;
2. in the case of a single array reference involved in this dependence, the number of sd-equivalence classes is trivially two, as the data space of the array

is divided in two from the overlap with the boundary, resulting into two corresponding partitions on the ID;

3. in the *streaming-oriented* graph construction, read nodes with the same array reference are merged;

4. hence, when there is more than one *cyclic-read* node, the preimage of the array reference of their respective data domains cannot completely overlap, as those references have *necessarily* different subscript functions, and nevertheless all are applied on the same ID.

That said, we can now define the number of sd-equivalence classes as:

Theorem 4.3.1. Number of sd-equivalence classes for a given cyclic-write node. *A given* cyclic-write *note* w *has a number of sd-equivalence classes* n *which is given by the following formula:*

$$n = 1 + k$$

where k *is the number of* cyclic-read *nodes. If the number of* cyclic-read *nodes is* $k = 0$*, there is only one sd-equivalence class, as indeed* n *is equal to just* 1*.*

However, determining the ID of each sd-equivalence class, as well as the set of actual input filters, is a completely different task, as it will require a specific algorithm. This algorithm employ the original ID of the *cyclic-write* node and the subscript function of each *cyclic-read* node. An important precondition for the applicability of the algorithm is that it operations of *intersection* and *difference* between polyhedra can be performed easily by employing the state of the art library *isl* [112], a library for manipulating sets and relations of integer points bounded by linear constraints.
The result is the set of sd-equivalence classes with the associated input filters and ID. Note that the *read* nodes which are not *cyclic-read* nodes are implicitly input of each sd-equivalence class, as they themselves are indeed implemented as a single filter.

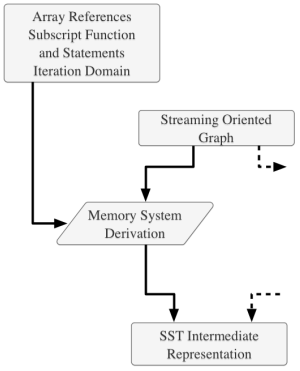
Algorithm 5 sd-Equivalence Classes Extraction

Input: I: the ID 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 is the symbolic value representing the part of the data domain which overlaps with I, while f_{nw} represents the part which overlaps with the boundary.
Output: E: set of equivalence classes $e = (i_e, r_e)$, where i_e is the ID and r_e the set of input f_w.

```
E ← 0
P ← 0 {P is the set of the preimage portion of each a_i that overlaps with I}
for all a_i ∈ A do
    D_{a_i} ← f_{a_i}
    S ← D_{a,i} ∩ I
    p_{a_i} ← (f_{a_i}^{-1}(S), a_i) {the first element of the tuple is the preimage, the second is the identifier}
    P ← P ∪ p_{a_i}
end for
i_0 ← ∩_i preImage(p_{a_i}), p_{a_i} ∈ P
E ← E ∪ e_0 = (i_0, F = {f_w(a_i) | ∀a_i ∈ A})
i_1 ← I − (∪_i p_i, p_i ∈ P)
if i_1 ≠ 0 then
    E ← E ∪ e_1 = (i_1, 0)
end if
while P ≠ 0 do
    p_{a_j} ← firstElement(P)
    Temp ← P − p_{a_j}
    i_new ← preImage(p_{a_j})
    r_new ← 0
    r_new ← r_new ∪ f_w(identifier(P_{a_i}))
    while Temp ≠ 0 do
        t_{a_k} ← firstElem(Temp)
        Temp ← Temp − t_{a_k}
        i_old ← i_new
        i_new ← i_new ∩ preImage(t_{a_k})
        if i_new = 0 then
            i_new ← i_old
        else
            r_new ← r_new ∪ f_w(identifier(t_{a_k}))
        end if
    end while
    for all p_{a_i} ∈ P do
        preImage(p_{a_i}) ← preImage(p_{a_i}) − i_new
        if preImage(p_{a_i}) = 0 then
            P ← P − p_{a_i}
        end if
    end for
    E ← E ∪ e = (i_new, r_new)
end while
```

The complexity of the proposed algorithm is $O(n^2)$ in the worst case, where n is the number of *cyclic-read* nodes. However:

- the worst case occurs only when the number n is *very* small (near to 1),
- in real world cases, n is always small enough for the algorithm to terminate in an affordable time.



4.3.3 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 splitted in two, and implemented as two different filters, for the previously described reason. 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 - *i.e.* whether it is the output of the *write-node* or not - to obtain the so called *chains*. In order to disambiguate, the different chain will be referred hereafter as:

   - *output-chain*, the one whose input stream is the *cyclic-write* node's output. This chain could be absent, whenever the *write* node is not a *cyclic* one.
   - *input-chain*, the other chain whose array is the same as the one updated by the *write* node.
   - the remaining will be simply referred as generic 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 and flow following the *reverse lexicographic order*, up to the minimum, which means that those filters are linked together by the input stream. Those links, which are indeed communication channels between the filters, will effectively implement the optimal

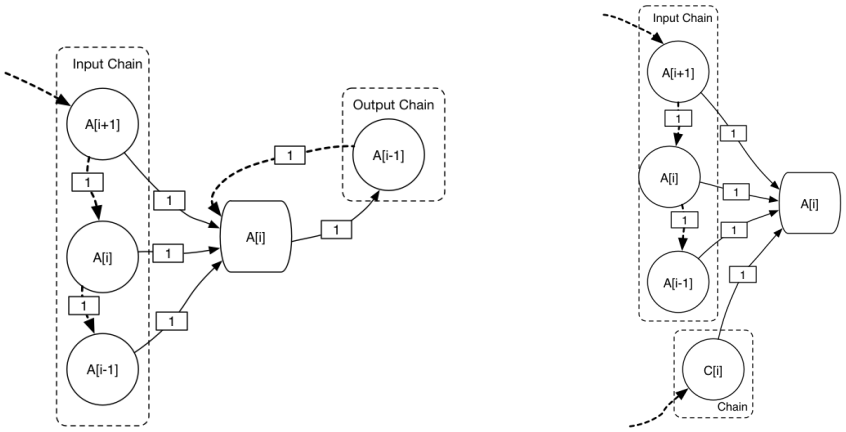
Full Buffering (FB), employing a non-uniform memory partitioning of the input stream. The channels size are in fact computed as the modulus of the *data distance vector* of a *fixed* and *common* - but nevertheless arbitrary - iteration between the subscript function of the filter which writes in the communication channel and the subscript function of the filter which reads from the communication channel.
The partitioning is non-uniform because the communication channels (the buffers) size will be non-uniform, shaped following the access patterns. The microarchitecture is optimal with respect to FB as those buffers will be both in the minimum possible number, given that all the array accesses can be done concurrently, and with the minimum possible size that can allow full data reuse. We chose a reverse lexicographic ordering of filters within the chain because, considering the fact the input stream arrives following the lexicographic order, the opposite ordering of the chain would make it not work properly. In fact, a lexicographic ordering of the chain would lead to have the first filter to start filtering data to the computation system before the others could even read from the input stream. As we employ, in order to have independent modules, blocking read and writes, the first filter would stall waiting for the computation to proceed, also stalling the input stream flow, and thus causing a deadlock. In summary, we can define the conditions for the proper structuring of a chain as:

Definition 4.3.3. *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 FB 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
$$W \geqslant | \delta(\nu, \nu)_{f_A^i f_A^j} |$$
If W is minimal ($=$), the FB is also *optimal*.

3. As stated before, there is also the boundary of the output array to consider. Within the *input-chain* the filter whose data domain perfectly overlaps with the ID, *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 eventually this node is not present (the stencil shape does not include the "central" point, *i.e.* the update of a grid point is performed without reading the 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 a chain, will be of size 1.



(a) *sample1*, listing 4.3. (b) *sample2*, listing 4.4.

Figure 4.10: The resulting chains for both samples.

4.3.4 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. This IR indeed 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 class:
  - an identifier
  - Its ID
  - The array update formula
  - The input communication channels, as well as the output one

- For each communication channel:
  - an identifier
  - Its minimum size

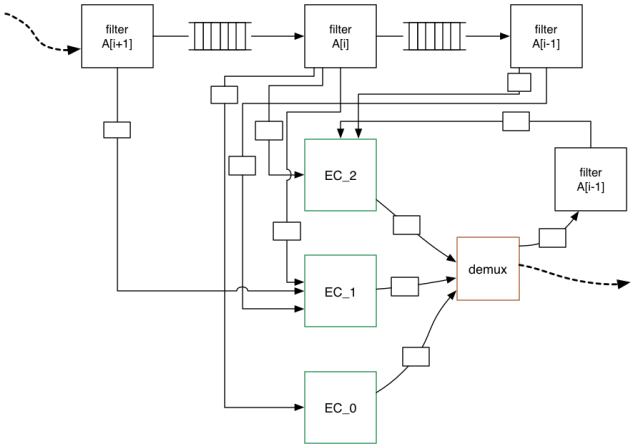
Information on the demux are not needed - they would be indeed redundant - as its structure can be inferred by the ID of the associated equivalence classes.

From the SST's IR the hardware equivalent is generated employing High Level Synthesis (HLS), hence, it is required to generate the code for each module of the microarchitecture: demuxes, equivalence classes and filters.The communication channels will be implemented as First In First Outs (FIFOs) queues.
The modules code can be easily generated using the state of the art tools of the PM, the most important of which is *CLooG* [3], 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. There are however two important remarks about the SST code generation:

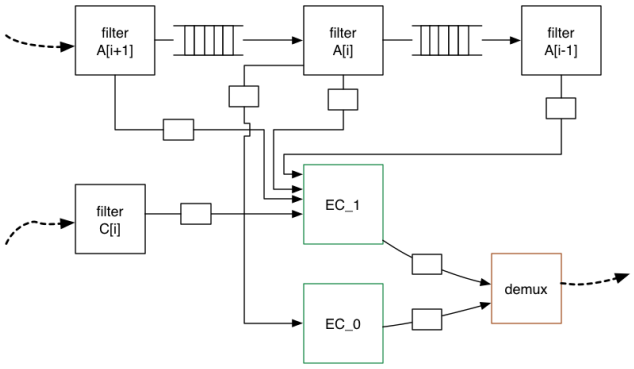
- The boundary transfer is made with a single channel from the filter to the demux. No additional modules will be inserted in between, as they are indeed unnecessary;

- The equivalence classes could in principle be implemented within one single module. As a matter of fact, doing so would force the code generation

phase to insert conditionals inside the loop nest. The presence of conditionals in the synthesized module can however lead to unnecessary control overhead, which can result in an overall slow-down or even in the impossibility to reach timing closure, considering that update formulae when synthesized may already themselves require a non negligible number of clock cycles. In order to avoid this situation, we decided to implement each equivalence class as a single module, with the aim of having generated code that is as simple as possible. Notice that whenever an equivalence class takes as input constants - but also array with *constant subscript function* $f(\vec{x}) = \vec{a}$ - the actual read instruction will be placed at the beginning of the generated code, before the loop nest, since they will not change during the computation, thus they are needed to be read only once.



(a) *sample1*, listing 4.3.



(b) *sample2*, listing 4.4.

Figure 4.11: A representation of the resulting SSTs for both samples.

4.3.5 Pipelining the SST

Once the chains are running steadily, they are able to send new data to the computation system within a *very small* number of clock cycles. It will be however very likely that the *equivalence classes* will not be able to produce new output at the same ratio, since there is a complexity gap between the synthesized circuit of an equivalence class and a filter, due to the fact that equivalence classes are demanded to perform the actual computation, while a filter's purpose is only to route the data stream, which result in a substantial difference in latency.
Hence, to increase the overall throughput it is quite obvious that equivalence classes must be speeded up. In a streaming system, pipelining the computation is a well suited technique that can effectively help to increase the overall throughput. Therefore, in our case a speed up could be obtained by pipelining the equivalence classes. This has proven to reduce the execution time by orders of magnitude, as will be shown in chapter 5.
However, care must be taken in using this type of optimization, because it could lead to situations of deadlock. If we indeed consider the fact that blocking reads and writes are used from every SST module, a pipeline of an equivalence class could stall because of the absence of input data, and thus produce no output. A stalled equivalence class could lead to block the computation whenever the demux is listening on its communication channel and the memory system has instead started to feed another equivalence class, that cannot proceed as its communication channel towards the demux will be full before the demux could even start to read from it, causing then a deadlock. This situation can be explained better with an example.

Let us consider an SST with two equivalence classes: the first being the boundary EC_0, the second being the computation part EC_1 with a pipeline of depth p. At the beginning of the computation, the demux D starts to read data from the memory system M through the communication channel of EC_0, and after a while begin to read on the communication channel with EC_1. Meanwhile, EC_1 reads the needed data from M, and since it is pipelined, it reads from M at every clock cycle, not producing however any output before p clock cycles has passed.

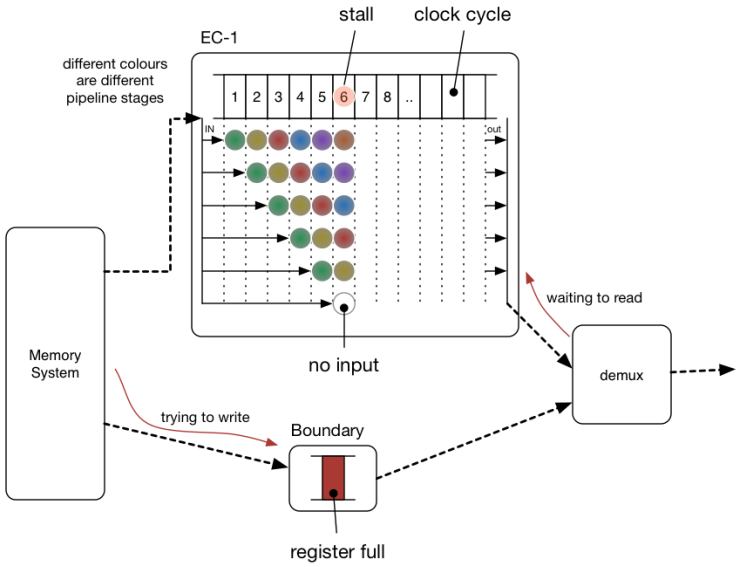


Figure 4.12: A deadlock condition occurred because EC_1 is fully pipelined.

M finishes to send data to EC_1 in $k < p$ clock cycles, and begin to send again data towards EC_0. The pipeline of EC_1 is then stalled. D is still waiting to read the first output of EC_1 on its communication channel, thus it will not take any data from the channel of EC_0. M soon fills the communication channel of EC_0 with D, and then stall, as it cannot proceed because it has no space to write. This is indeed a deadlock, as M, EC_1 and D are all stalled and cannot proceed.

There are two solutions to this problem:

- The communication channels between the memory system and any equivalence class could be oversized, thus allowing the memory system to proceed even if the equivalence classes are stalled. However, this solution has two important drawbacks. The first is that if those channel are oversized, than the SST will not anymore enjoy the property of having optimal FB. The second and most important is that to compute the communication channels size the pipeline depth of each equivalence classes must be known, something that in general is not easy to do.

- Only the innermost loop of each equivalence class is pipelined, which result into the flushing of the pipeline right when the memory system starts to send data to another equivalence class. This hence allows the production

of the output of the equivalence class, that cannot stall anymore, avoiding completely the possibility of a deadlock within an SST. This solution will be further detailed in chapter 5.

There is a last, important, remark to be made. In the case of the presence of spatial dependencies, the related equivalence classes *cannot* be pipelined, as the input of some of them will depend from their output, recombined by the demux, resulting in a cyclic dependency that will surely lead to a situation of deadlock. This is a structural limitation, and cannot be resolved in any way.

4.3.6 Scaling on the Problem Size

As already stated in section 3.3.1, 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. In practice, this means that there is always the possibility to remove the *largest* communication channel and 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 - obviously, though - by the available off-chip bandwidth.

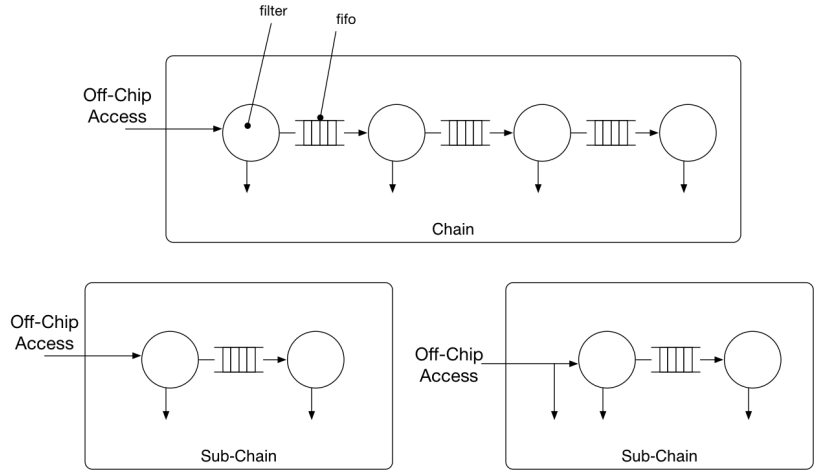
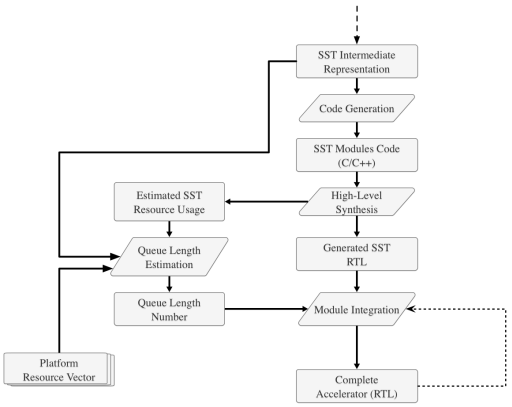


Figure 4.13: A communication channel is removed, to reduce the memory space requirements, and substituted with another off-chip access.

As last observation, although it could lead to a decrease of the overall performance, some FIFO channels could be implemented as distributed SRAM FIFOs (LUT-based) instead of BRAM-based, in order to balance the usage of the different components. This could be another solution to allow the scaling on the problem size.

4.4 The SSTs Queuing Technique



The functionality of an SST is, as already stated, to perform the computation within a single time-step. Hence, having an hardware accelerator made 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 already seen issue for ISLs, their inherent memory boundedness is in fact the reason why obtaining high performance with ISLs is in general a hard 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. The SSTs *queuing* is exactly such a technique.

Its key point is that multiple SSTs are arranged in a queue fashion, which means that, within the queue, the output of one SST is the input of the next. Off-chip memory transfers occurs then *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 amount* of off-chip/on-chip memory transfers, which in turn

means that the bandwidth requirements *will remain constant*. Therefore, the memory boundedness degrades progressively as the queue length increase, since a greater volume of computation will be performed with the same bandwidth requirements.

The queuing technique takes its advantage from a peculiarity of the single SST: its streaming-based computation. In fact, the continuous data flow ensured by a streaming system allows, within this context, to have a *pipelined* computation in the SSTs queue. As a matter of fact, the SST can be viewed as a set of Processing Elements (PEs) connected in series, each operating on a "portion" of the stream, *i.e.* as soon as an SST produces data, this data will flow into the queue in order to be processed by the subsequent SSTs. A streaming-based computation of an SST allows then to have, at a certain point, a concurrent processing of *all* the SSTs, therefore, as previously stated, a *pipelined* computation.

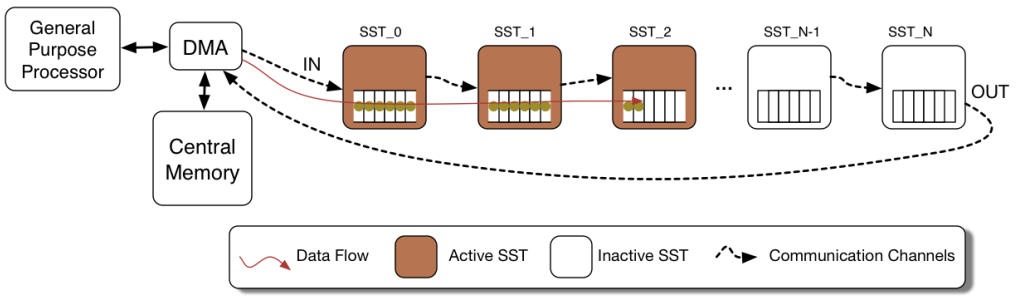


Figure 4.14: A visualization of the pipelined execution within the queue.

We claim that employing the SSTs queuing will *speed up the ISL 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:

$$C_q = \frac{T}{q} * (N * (m_{in} + q * sst + m_{out}))$$

Consider now $m_{in} + m_{out}$ to be equal to sst multiplied by a certain constant $k \in \mathbb{R}$, hence:

$$m_{in} + m_{out} = k * sst$$

the previous two formulae now become:

$$C = T * (N * ((k+1) * sst)) \quad \text{and} \quad C_q = \frac{T}{q} * (N * ((k+q) * sst))$$

If $k \gg q$, which in turn means $k \gg 1$, as obviously $q > 1$, we can perform these approximations:

$$k + 1 \approx k \quad \text{and} \quad k + q \approx k$$

Therefore:

$$C \approx T * (N * k) \quad \text{and} \quad C_q \approx \frac{T}{q} * (N * k)$$
$$C_q \approx \frac{C}{q}$$

Even without the approximation made above, the same proof would still hold whenever T or N are large numbers. Indeed, if:

$$T * N \gg q \text{ and } T * N \gg m_{in} + sst + m_{out}$$

it would mean that:

$$T * N * (m_{in} + sst + m_{out}) \approx T * N * (m_{in} + q * sst + m_{out})$$

hence, as before:

$$C \approx T * N \quad \text{and} \quad C_q \approx \frac{T*N}{q}$$
$$C_q \approx \frac{C}{q}$$

□

Remark. The speedup is pseudo-linear because of the approximation $k + q \approx k$ made in the first case, or of the approximation $T*N*(m_{in}+sst+m_{out}) \approx T*N*(m_{in} + q * sst + m_{out})$ made in the second. Furthermore, the effect of queuing, whenever T and N are small, could in principle be mitigated if the queue length q reaches the same order of magnitude of k, which, however, is a difficult condition

to be achieved, considering the possibility of *queue looping*, a concept that will be explained in a moment.

The proof is also not exact, as indeed it assumes implicitly in the first case that the transfer time is much greater than the time spent within an SST, $m_{in} + m_{out} \gg sst$, which is however true in most cases, as it will be empirically proven in chapter 5, or in the second case that T and N are large numbers, which is also quite common in real ISLs.

4.4.1 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 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 of time-steps of the ISL.

By employing this information, the estimation process can actually be represented as simple division between R_{max} and the sum of the needed resources for both the communication channels - also those necessary to forward data from one SST to the other - and the SST, limited however, whenever the number of time-steps t is relatively small, by the obvious constraint 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 4.4.1. Q_{max} *estimation*. An SST holds a fraction f of the sum of all the array 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:

$$Q_{max} = \min\{q \mid \sum_q f > S_A \,,\, q \in \mathbb{N}\}$$

In order to allow the queue looping, we already stated in section 3.3.2 that a *mux* must be added to the accelerator (figure 3.1 (c)), hence when estimating the total resource usage, the presence of the mux must be taken into account. This also happens when the queue length is not an exact divisor of the total number of time-steps, as the mux will break the queue when the total number of time-steps of the ISL are executed (figure 3.1 (b)). Notice that a mux is indeed very similar to a filter, since its functionality is simply of routing the streams. Therefore, generating the code for a given mux can be made in the exact same way as of a filter.

Lastly, we recall, as shown in figure 4.1, 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 until the accelerator fits onto the available resources.

4.4.2 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 3.3.2 (figure 3.4). 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.

There is however an additional remark that must be done when dealing with

more than one input array, *i.e.* the possibility of deadlock when pipelining is enabled within an SST. This is indeed an extension of the problem already analyzed in section 4.3.5. Since an SST is a streaming-based microarchitecture, the chains which have the added communication channels will be forced to forward the data to the next SST as they read it, which by the way corresponds to the moment in which they forward it to their computing system. This means that those chains can stall whenever the next SST does not empty these communication channels, which by the way are of size 1. In order to explain how the deadlock can occur, let us now consider two SSTs, S_1 and S_2, S_2 being the subsequent in the queue. both S_1 and S_2 are pipelined. S_1 will not produce any output - except the boundary - until the first equivalence class E_1 has not computed data to be forwarded to its demux D. E_1 reads from a chain C_1 that forwards also the data to the chain C_2 of S_2. S_2 is however stalled waiting for the stream of D. the communication channel between C_1 and C_2 is filled within few clock cycles, as is indeed of size 1, causing C_1 to stall trying to write onto a full channel. The fact that C_1 is stalled will cause also E_1 to stall, as it cannot proceed until C_1 does not provide the needed input. This overall stall cannot be recovered, causing a deadlock.

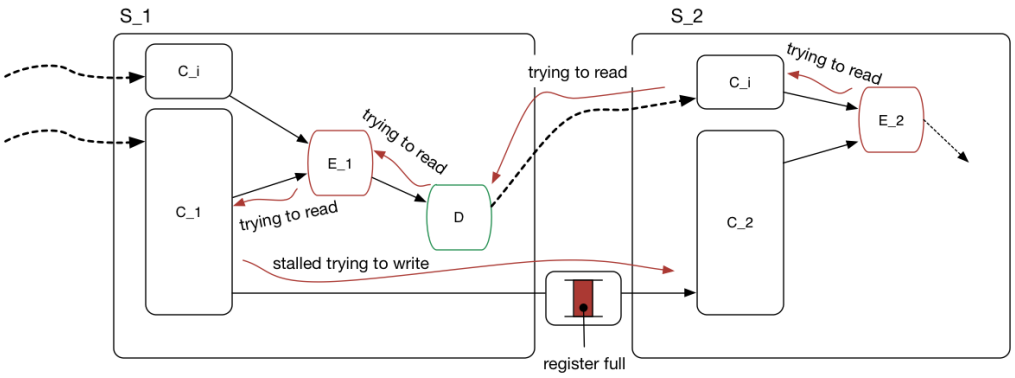


Figure 4.15: The deadlock condition that can occur when queuing pipelined SSTs with multiple inputs.

The solution to this problem is to oversize *only* the added communication channels - the one demanded to forward the multiple inputs - between SSTs, in order to allow the related chain to proceed even if the subsequent SST is actually stalled. The size of these communication channels can be computed as the largest *iteration vector* size among all the innermost loops of an rSCoP, given that however *only* the inner loop of each equivalence class is pipelined, as described

in section 4.3.5.

5

Results

The focus of this Chapter is to prove the validity of the proposed accelerator. The experimental results are presented in Section 5.3, preceded by the description of both the experimental setup in Section 5.1, and the used benchmarks in Section 5.2.

5.1 Experimental Setup

To test the proposed hardware accelerator we employed the *Vivado Design Suite* [11]. The Streaming Stencil Time-step (SST) microarchitecture derivation has been partially aided by state of art polyhedral tools, and partially done by hand. We performed the polyhedral analysis phase with:

- *Clan [2] (Chunky Loop ANalyzer)*, to extract a polyhedral Intermediate Representation (IR) from the source code;
- *Candl [1] (Chunky ANalyzer for Dependencies in Loops)*, to compute polyhedral dependencies, and thus the corresponding Data Dependency Graph (DDG), from the polyhedral IR.

The SST's modules have been implemented using *Vivado HLS* (v2014.3.1). This tool enables implementing the modules with C language and exporting the corresponding Register-Transfer Level (RTL) as a Vivado's Intellectual Property (IP)

core. The reader may refer to section 2.3 for further details about High Level Synthesis (HLS).

Both SST's modules integration and queuing have been performed using *Vivado* (v2014.3.1), used also 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 to push queuing only to a fraction of the total available resources, since beyond a given percentage of the total area we systematically ran out of memory during place and route.

All the tests have been performed on the VC707 board, which has a Xilinx Field Programmable Gate Array (FPGA) chip *Virtex-7* XC7VX485T. Along with other resources, the board features a 1GB DDR3 RAM, which we also employed in our tests as reference off-chip memory.

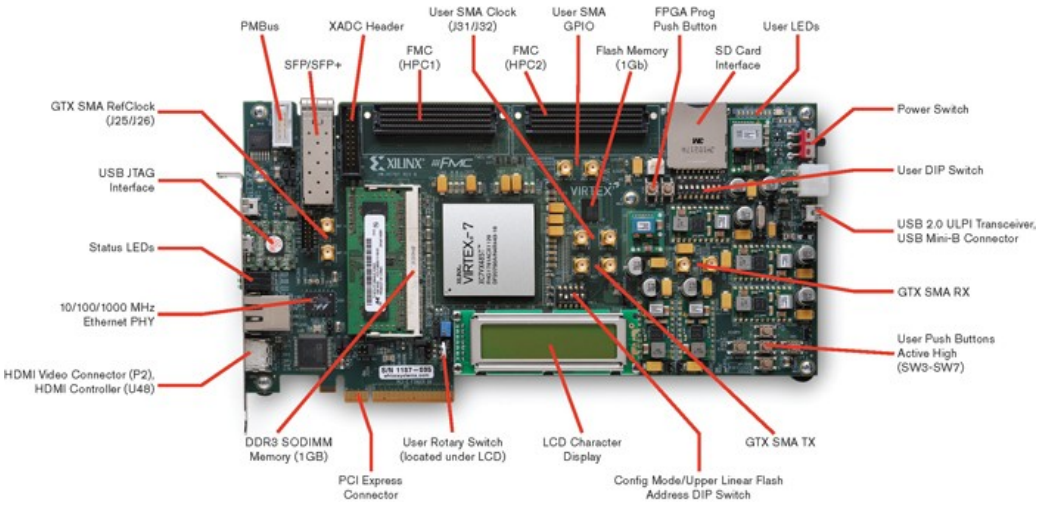


Figure 5.1: The VC707 board. Image taken from the product site.

In order to perform the actual testing, we have also implemented onto the Virtex-7 an embedded system, mainly composed of the following parts:

- An host processor, the *MicroBlaze*, a soft processor core designed for Xilinx FPGAs. Since it is a soft processor, it is entirely implemented within the fabric of the Virtex-7. To program it and hence drive the whole testing phase, we used *Vivado SDK* (v2014.3.1).

- A memory interface towards the 1GB DDR3, implemented through a *mem-*

ory interface generator, to which we attached one or more Direct Memory Accesses (DMAs) according to the needs of the specific test case.

- A *timer*, thanks to which we have measured the total number of clock cycles required by the accelerator.

The operating frequency we used to generate the HLS modules, and to which we put the entire system, in all our tests, is *200MHz*.

5.2 Test Cases

Let us now focus on the three benchmarks we chose to test our accelerator. Two of them belong to *PolyBench/C* [8], which is a collection of benchmarks containing Static Control Parts (SCoPs), designed to test polyhedral based optimizations, and are *jacobi-2D* and *seidel-2D*. The third is taken from the FASTER's [4] implementation of Reverse Time Migration (RTM), and is the function *do_step*. The rationale beyond these choice is that the three benchmarks are three relevant case studies, as indeed:

- *jacobi-2D* is the simplest of the three, but due to its nature it is well suited to perform 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 *jacobi-2D*, we were still able to push queuing up to a considerable number of SSTs without running out of memory during the synthesis process.

- RTM's *do_step* is a 3-dimensional, compute- and memory-intensive Iterative Stencil Loop (ISL) with variable coefficients, and thus multiple input arrays;

- *seidel-2d* contains spatial dependencies between grid points updates, hence it has no parallelization opportunities.

All three benchmarks have been carried out with single precision *floating point* data types.

5.2.1 Polybench/C Jacobi 2-D

```
1 for (t = 0; t < T_Step; t++){
2     for (i = 1; i < N-1; i++)
3         for(j = 1; j < M-1; j++)
4             B[i][j] = 0.2*(A[i][j]+A[i][j-1] + A[i][1+j] + A[1+i][j] + A[i-1][j]);
5     for(i = 1; i< N-1; i++)
6         for(j = 1; j < M-1; j++)
7             A[i][j] = B[i][j];
8 }
```

Listing 5.1: Polybench/C Jacobi 2-D

The Jacobi method is a popular algorithm for solving Laplace's differential equation on a square domain, regularly discretized. As can be seen from listing 5.1, it is a 5-points ISL, and within a single time-step it does not have spatial dependencies between updates of points. The number of Floating Point Operations (FLOPs) for the update statement, line 4, is 5, being 4 sums and a multiplication. For this benchmark, we used a problem size where $N = 1080$ and $M = 1920$, thus being a FULL-HD array.

5.2.2 FASTER RTM 3-D

RTM is a powerful seismic imaging method for the interpretation of steep-dips and subsalt regions. Since the original implementation of the FASTER's version of RTM was not suitable to be successfully analyzed by our methodology, we focused on the only *do_step* function, the one responsible for the actual stencil update. We slightly modified it to be used as an effective benchmark, and the result is presented in listing 5.2. It is a 31-points, 3-dimensional ISL, and it also employs multiple arrays during the computation. In this case the number of FLOPs of each array update is 51. The problem size we chose is 100x100x100, being the first of the three used by the FASTER's implementation itself.

```
for (t = 0; t < T_Step; t++){
    for(i=5; i < N-5; i++){
        for(j=5; j < N-5; j++){
            for(k=5; k< N-5; k++){
                pp[i][j][k]=(2.0*p[i][j][k]-pp[i][j][k]+dvv[i][j][k]*(
                            p[i][j][k]*c[0]
                            +c[1]*(p[i+1][j][k]+p[i-1][j][k])
                            +c[2]*(p[i+2][j][k]+p[i-2][j][k])
                            +c[3]*(p[i+3][j][k]+p[i-3][j][k])
                            +c[4]*(p[i+4][j][k]+p[i-4][j][k])
                            +c[5]*(p[i+5][j][k]+p[i-5][j][k])
                            +c[6]*(p[i][j+1][k]+p[i][j-1][k])
                            +c[7]*(p[i][j+2][k]+p[i][j-2][k])
                            +c[8]*(p[i][j+3][k]+p[i][j-3][k])
                            +c[9]*(p[i][j+4][k]+p[i][j-4][k])
                            +c[10]*(p[i][j+5][k]+p[i][j-5][k])
                            +c[11]*(p[i][j][k+1]+p[i][j][k-1])
                            +c[12]*(p[i][j][k+2]+p[i][j][k-2])
                            +c[13]*(p[i][j][k+3]+p[i][j][k-3])
                            +c[14]*(p[i][j][k+4]+p[i][j][k-4])
                            +c[15]*(p[i][j][k+5]+p[i][j][k-5])
                            ))+source_container[i][j][k];
            }
        }
    }

    for(i=5; i < N-5; i++)
        for(j=5; j < N-5; j++)
            for(k=5; k< N-5; k++)
                p[i][j][k]= pp[i][j][k];
}
```

Listing 5.2: Modified *do_step* of FASTER's RTM 3-D

5.2.3 Polybench/C Seidel 2-D

```
for (t = 0; t < T_Step; t++){
    for (i = 1; i < N-1; i++)
        for(j = 1; j < M-1; j++)
           A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
           + A[i][j-1] + A[i][j] + A[i][j+1]
           + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
}
```

Listing 5.3: Polybench/C Seidel 2-D

The Gauss-Seidel method is an iterative method used to solve a linear system of equations. The corresponding PolyBench/C's version, presented in listing 5.3, is a 9-points ISL that has spatial dependencies between points. In fact, since each update of an iteration depends upon all previously computed points, the updates

cannot be done simultaneously as in the Jacobi method, enforcing a sequential execution. The number of FLOPs of each array update is 9, being 8 sums and a division. As for *jacobi-2D*, for this benchmark we used a problem size where $N = 1080$ and $M = 1920$, thus being also in this case a FULL-HD array.

5.3 Experimental Results

In this section we present the experimental results of the previously described benchmarks. Before actually presenting them, there is an important remark that have to be done. In the first two benchmarks we experimented that the obtainable performance was bounded by the embedded system we employed. Indeed, since the datapath towards the off-chip memory was 32 bits wide, and the frequency 200MHz, the available bandwidth was 800MB/s. The estimated clock cycles of both the pipelined versions - we are referring to the single SST test - of jacobi-2D and RTM was $cc \approx 2$millions, but the actual measurement showed a total number of clock cycles that was $cc \approx 4$millions. This performance gap is certainly caused by the available bandwidth, as in both cases, considering the employed problem sizes, the total transfer would exactly take a number near 4 millions of clock cycles, indeed $cc_{total} = \frac{ProblemSize*Frequency}{Bandwidth}$.

This however does not influence the quality of the obtained results, as in fact our purposes when performing the tests were:

- to show how 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 synthesizing directly the original code via HLS;
- to show how the scalability given by the SSTs queuing ensure a pseudo-linear increase in throughput, while remaining with constant bandwidth, which is especially true in our case where the bandwidth was bounding the obtainable performance.

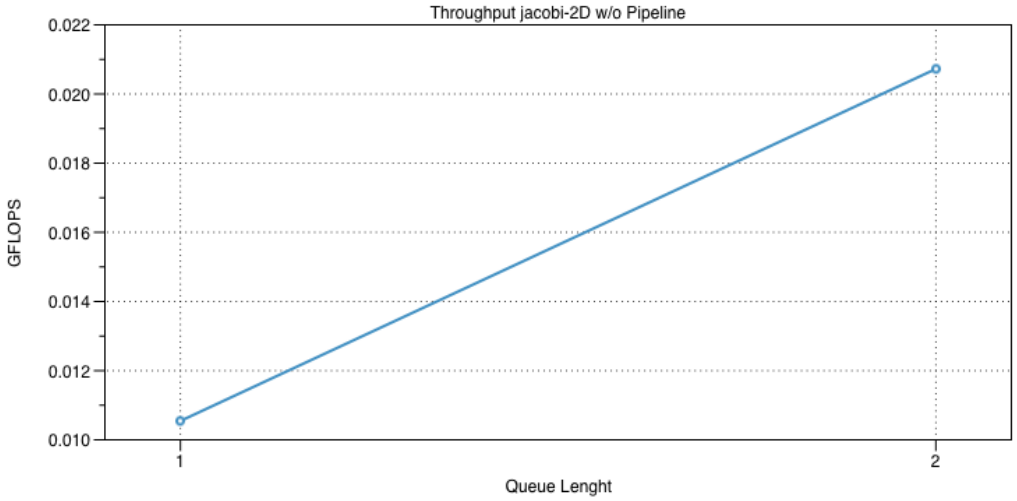
We compared our results with the Central Processing Unit (CPU) version, carried out on an Intel Core i7 2675QM with 8GB DDR3-1333 RAM. For the first two

benchmarks, we also performed a comparison with an implementation on Maxeler's MaxWorkstation. We chose the first comparison to show how employing an heterogeneous system can be highly effective with certain kinds of computations - ISL being one of them, and the second simply because the resulting architecture of our work is in a sense related to the dataflow architecture obtainable with Maxeler, as previously stated in section 3.3.4. Notice that however the bandwidth issue should be considered when comparing our results with the CPU and the MaxWorkstation, where the available bandwidth is higher than ours. A part from the fact that our goal was to prove the two aforementioned points, and not to compare our results, since it would be unfair due to the described bandwidth issue, there are two reasons why we did not make any comparison with the other existing works that, as far as we know, target ISLs - or even generic Static Affine Nested Loop Programs (SANLPs) - on similar FPGAs and employ the same benchmarks, and we instead chose to compare our results with actual measurements:

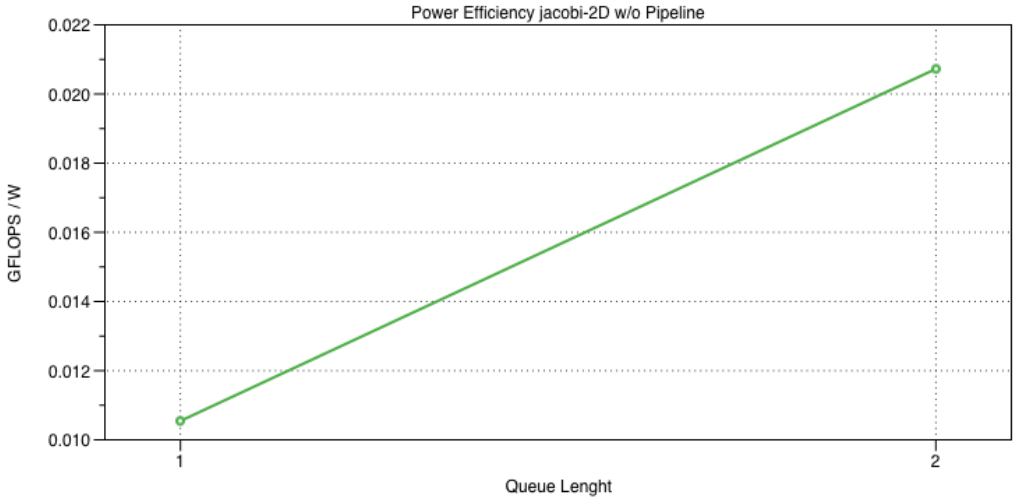
- All the works we inspected during our research did not provide comparable results, since they usually only provide the speedup with respect to a naive implementation, or simply they do not specify the workload (*e.g.* the problem size) which are essential if a comparison have to be made.
- Even worse, almost all of them provide only the estimated results (the HLS reports), they did not perform any real measurement on real hardware. This is indeed a huge limitation, as in these estimations potential deviations from the theoretical peak due to hardware constrains, such as the available bandwidth, are not considered.

5.3.1 jacobi-2D

Figure 5.2 shows the performance of the jacobi-2D version when pipelining within the SST is not enabled. Notice that, since the interesting benchmarks were those with pipelining enabled, we chose to test a queue length of only up to 2 SSTs, just to show how the throughput doubles passing from a single SST with a queue of length 2.



(a) Throughput (expressed in GFLOPS)

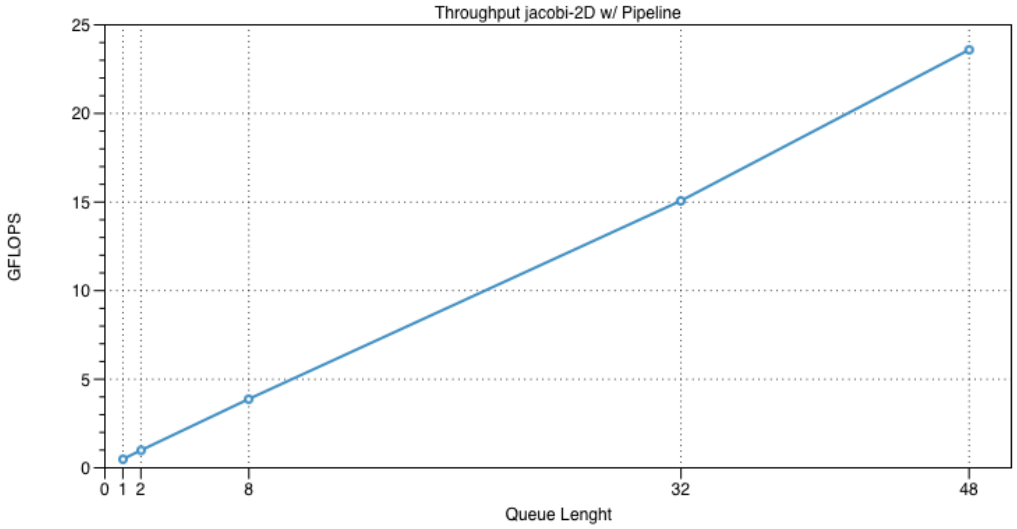


(b) Power Efficiency (expressed in GFLOPS/W)

Figure 5.2: Performance measurement of jacobi-2D without pipelining enabled within the SST.

In figure 5.3 the performance of the pipelined version of jacobi-2D is reported. In the case of jacobi-2D, we were able to synthesize up to 48 SSTs without running

out of memory during the synthesis process. Notice how in this case the pseudo-linear increase in throughput is even clearer than with the no-pipelining version. Observe also that with 48 SSTs the accelerator has a power efficiency greater than the top green500 system [5], which is of 5.271 GFLOPS/W (although in our case we are using single precision floating point, while the green500 measurements are done with double precision floating point). We expect to go even beyond this power efficiency systematically with the increase of the queue length, in all cases, due to the experimented linear scalability the SSTs queuing is able to deliver.

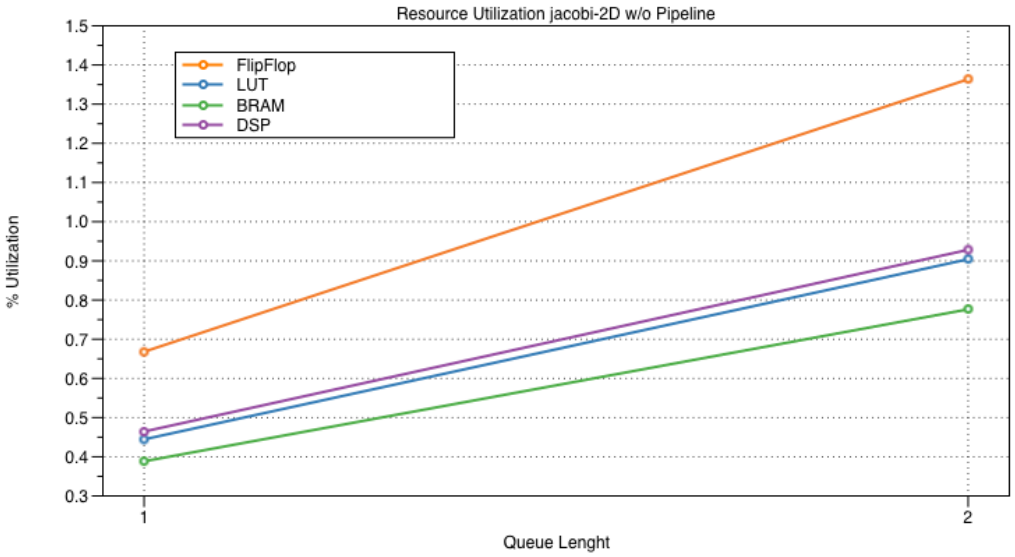


(a) Throughput (expressed in GFLOPS)

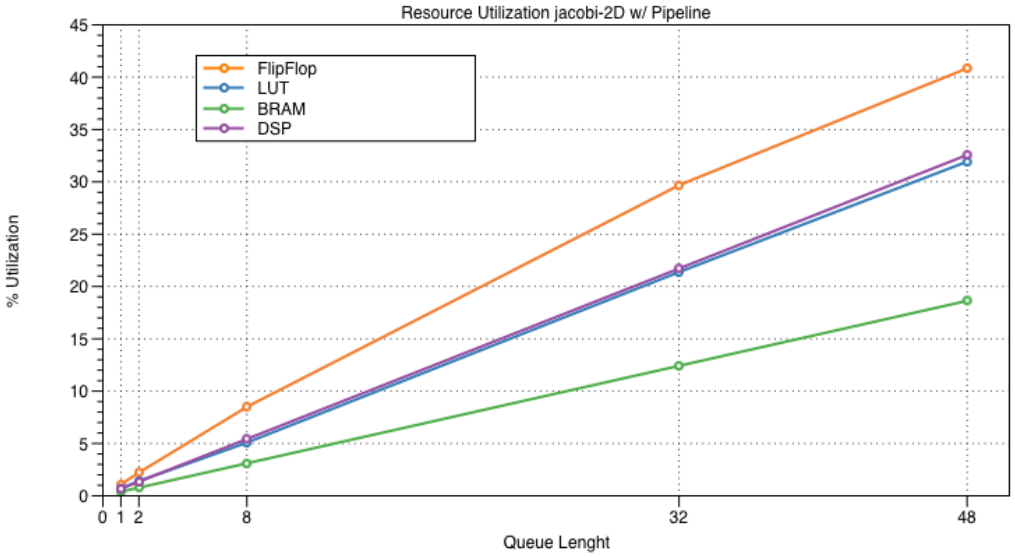
(b) Power Efficiency (expressed in GFLOPS/W)

Figure 5.3: Performance measurement of jacobi-2D with pipelining enabled within the SST.

Figure 5.4 shows the resource usage of all the performed tests, expressed as a percentage of the total available resources.



(a) without pipelining enabled within the SST



(b) with pipelining enabled within the SST

Figure 5.4: Resource usage of the accelerator for the jacobi-2D benchmark.

Table 5.1 summarizes the results of all tests performed with jacobi-2D, and reports also the results of the CPU and the MaxWorkstation.

Table 5.1: jacobi-2D

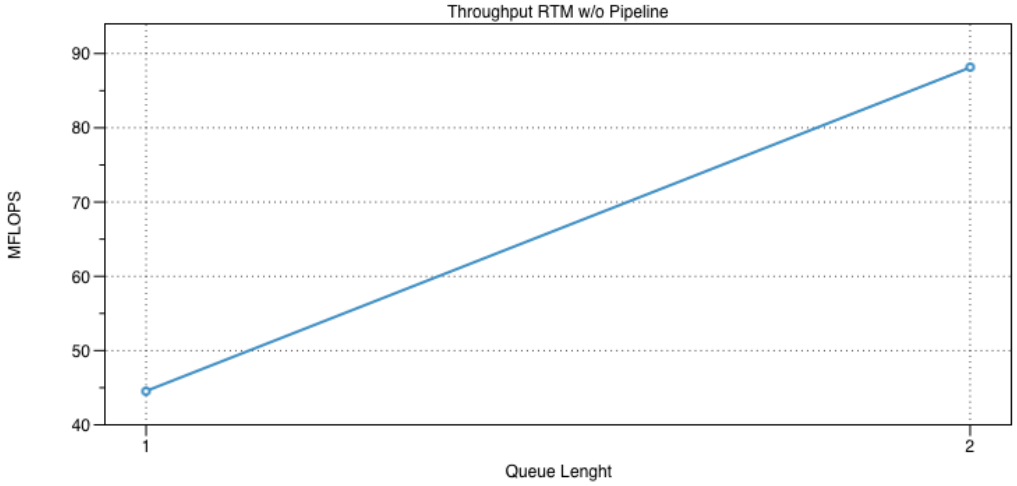
	Throughput	**Power Efficiency**	**LUT**	**FF**	**DSP**	**BRAM**
	(GFLOPS)	(GFLOPS/W)	(%)	(%)	(%)	(%)
CPU Sequential	1.662	0.066	-	-	-	-
CPU 8 Threads	6.688	0.268	-	-	-	-
MaxWorkstation	0.449	-	-	-	-	-
Naive	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
1 SST w/o Pipeline	0.030	0.011	0.444	0.668	0.464	0.388
2 SSTs w/o Pipeline	0.061	0.021	0.904	1.364	0.929	0.777
1 SST	0.490	0.171	0.696	1.089	0.679	0.388
2 SSTs	0.987	0.337	1.408	2.216	1.357	0.777
8 SSTs	3.887	1.191	5.083	8.500	5.429	3.107
32 SSTs	15.074	4.051	21.376	29.680	21.714	12.427
48 SSTs	23.596	5.775	31.931	40.847	32.571	18.641

Notice that the naive implementation, *i.e.* the one consisting of a direct synthesis of the original source code via HLS, could not fit on the board since the total available Block RAM (BRAM) is not enough to hold the entire involved data.

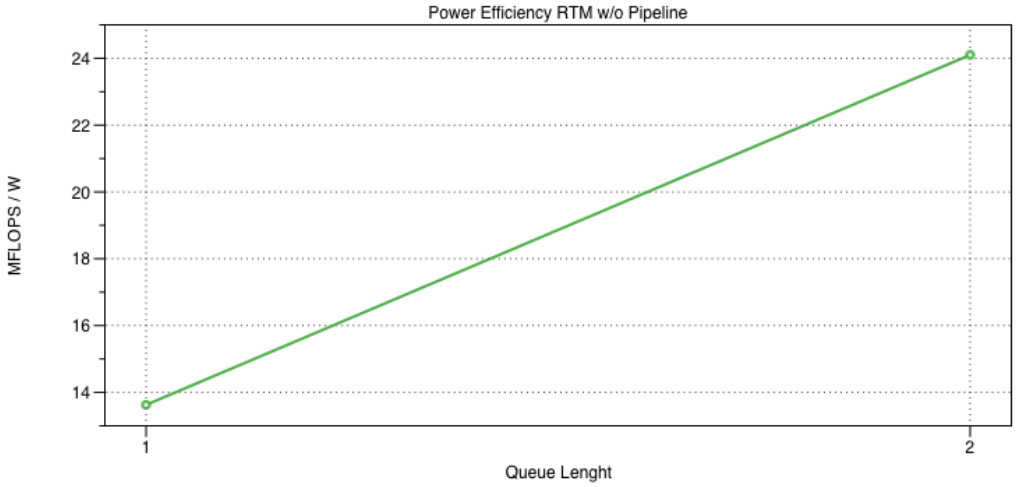
In the case of jacobi-2D, due to the absence of spatial dependencies within points update, a strongly hand-tuned parallel version have been also tested on CPU, where all the eight available cores have been employed.

5.3.2 RTM do_step

As for jacobi-2D, the first results are for the version without pipelining, shown in figure 5.5.



(a) Throughput (expressed in MFLOPS)

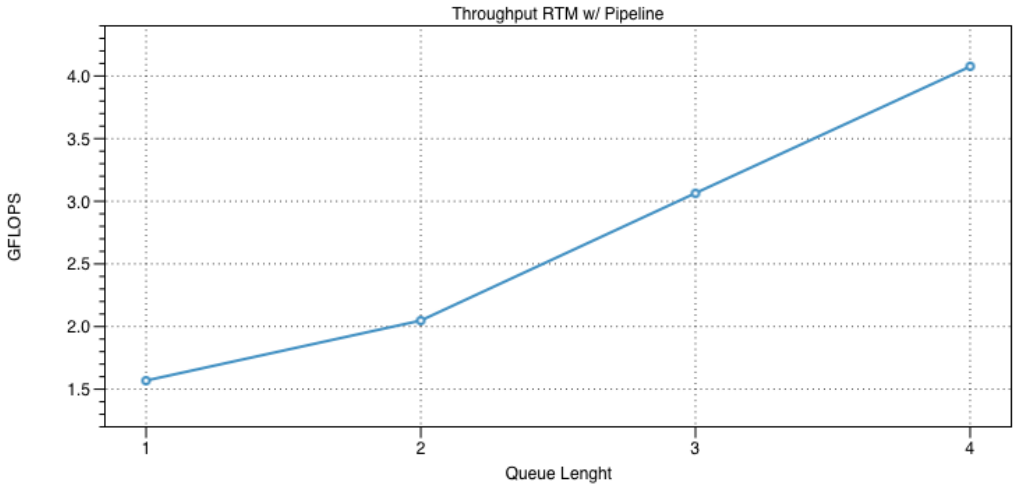


(b) Power Efficiency (expressed in MFLOPS/W)

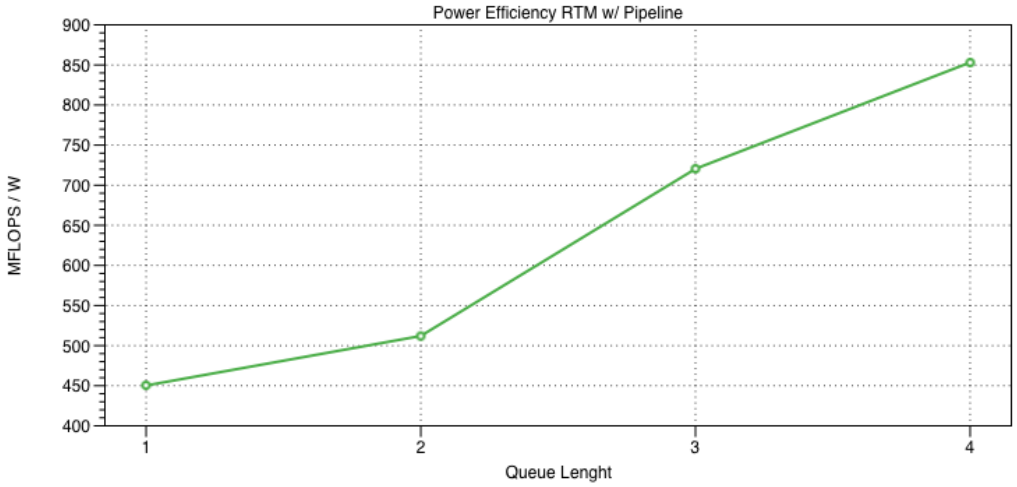
Figure 5.5: Performance measurement of RTM do_step without pipelining enabled within the SST.

In figure 5.6 we instead report the performance of the pipelined version of RTM do_step. Notice that we could not synthesize more than 4 SSTs without running out of memory during the synthesis process. The graphs shows however a linear increase in throughput even in this case. Due to the fact that this ISL employs multiple arrays in the computation, when performing queuing two different versions of the SST have to be used (as described in section 4.4.2): the

first has a single output - the array updated by the ISL - and is placed at the end of the queue, the second also routes the other involved streams, and is the one replicated within the queue. The second version is responsible to perform more operations, namely the forwarding of the other streams, hence it is a bit slower. For this reason, the throughput from 1 SSTs, where there is only the first SST version, to 2 SSTs, where both versions are in place, does not exactly double. From 2 SSTs onwards, since increasing the queue length consists in the replication of the second version of the SST, the trend is instead as expected.



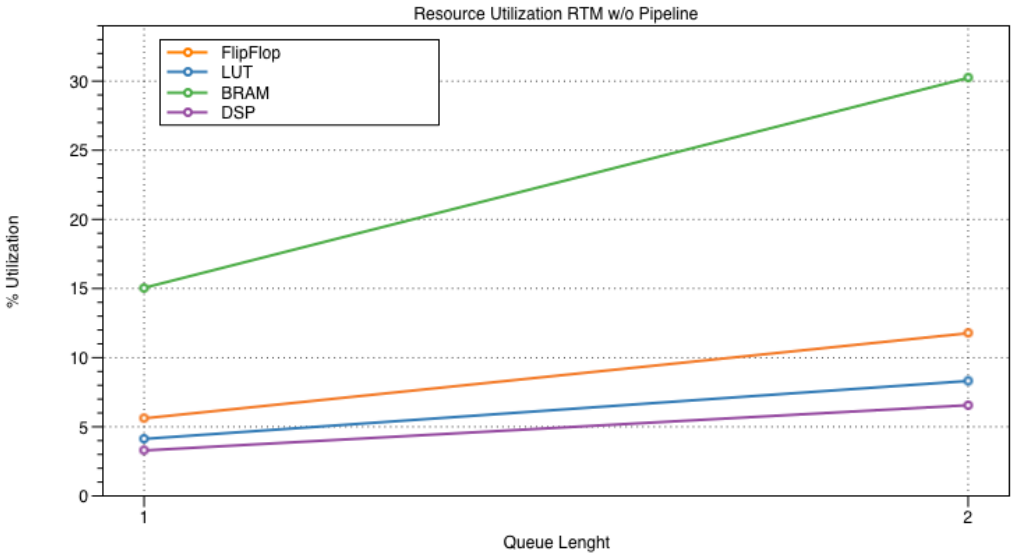
(a) Throughput (expressed in GFLOPS)



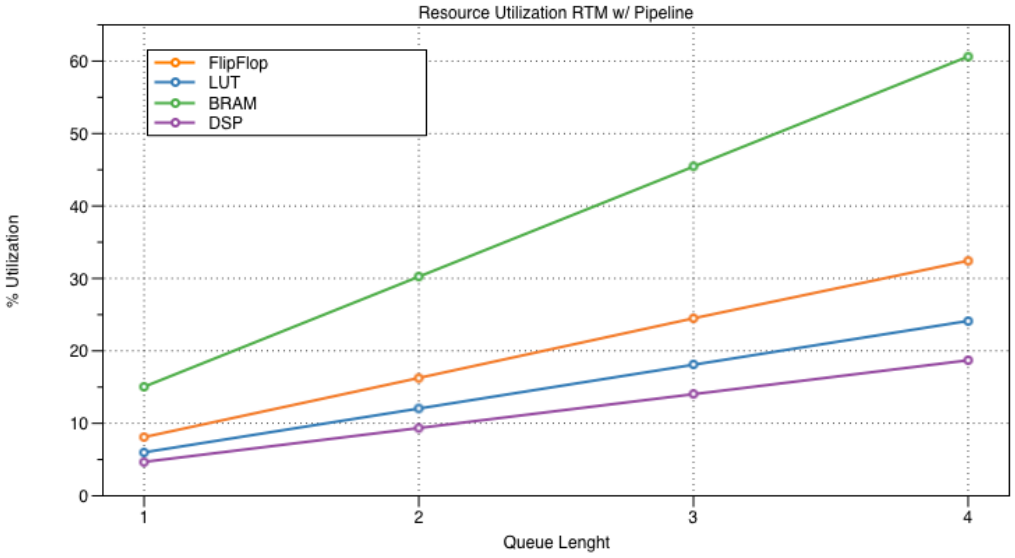
(b) Power Efficiency (expressed in MFLOPS/W)

Figure 5.6: Performance measurement of RTM do_step with pipelining enabled within the SST.

In figure 5.7 we report the resource usage of all the performed tests, as for jacobi-2D.



(a) without pipelining enabled within the SST



(b) with pipelining enabled within the SST

Figure 5.7: Resource usage of the accelerator for the RTM do_step benchmark.

Table 5.2 summarizes the results of all tests performed with RTM do_step, and reports also the results of the CPU and the MaxWorkstation.

Table 5.2: RTM do_step

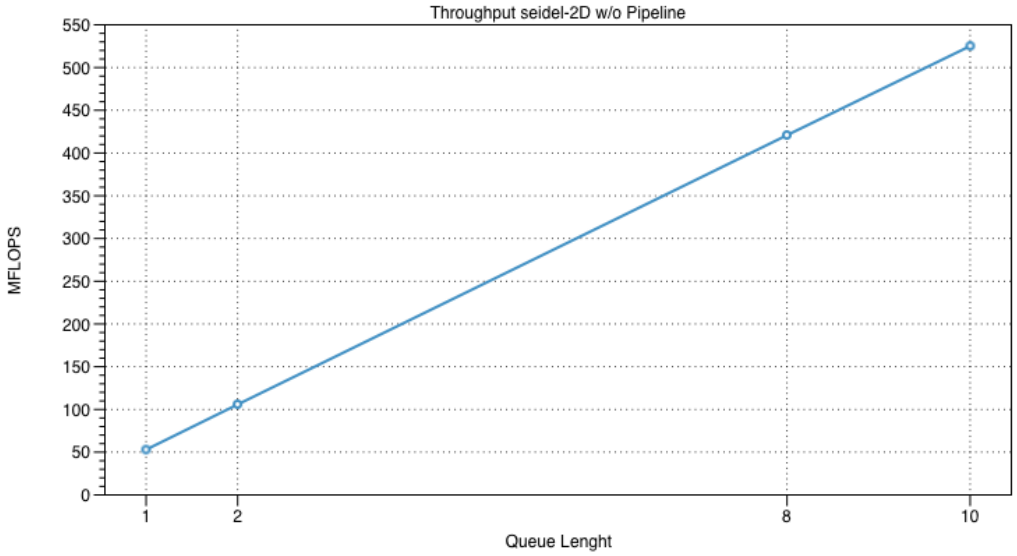
	Throughput	**Power Efficiency**	**LUT**	**FF**	**DSP**	**BRAM**
	(GFLOPS)	(GFLOPS/W)	(%)	(%)	(%)	(%)
CPU Sequential	2.271	0.091	-	-	-	-
CPU 8 Threads	5.084	0.203	-	-	-	-
MaxWorkstation	3.138	-	-	-	-	-
Naive	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
1 SST w/o Pipeline	0.045	0.014	4.122	5.609	3.286	15.049
2 SSTs w/o Pipeline	0.088	0.024	8.320	11.776	6.571	30.243
1 SST	1.569	0.450	5.996	8.100	4.679	15.049
2 SSTs	2.048	0.512	12.046	16.267	9.357	30.243
3 SSTs	3.065	0.720	18.095	24.494	14.036	45.437
4 SSTs	4.075	0.853	24.139	32.448	18.714	60.631

Even in this case, the naive implementation could not be implemented as it exceeds the total available on-chip memory requirements.

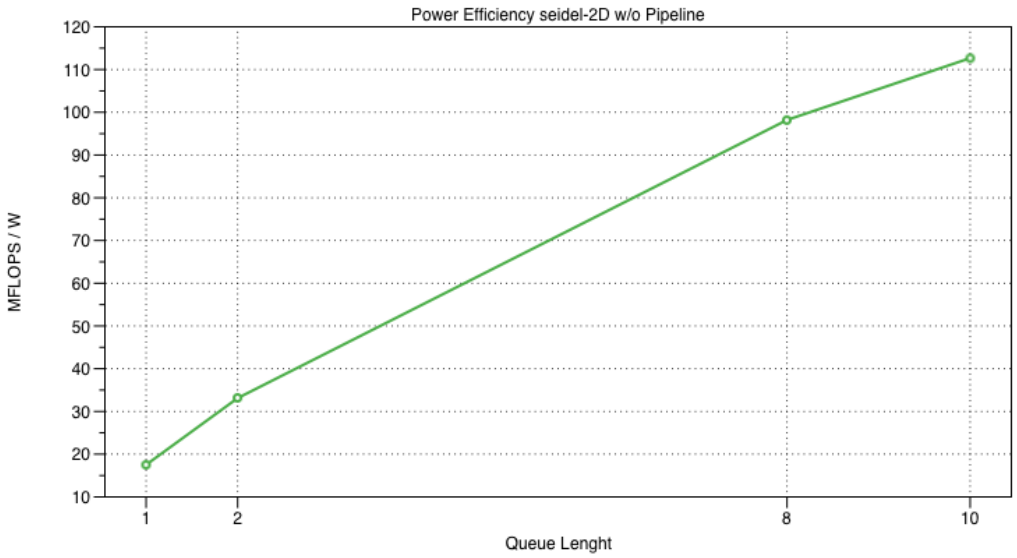
As for jacobi-2D, the absence of spatial dependencies has allowed to perform tests also on a strongly hand-tuned parallel version on the CPU.

5.3.3 seidel-2D

This ISL has spatial dependencies within points updates, thus a pipelined version cannot be obtained. Hence, only a no-pipeline version has been tested, for which performance and resource usage are reported respectively in figure 5.8 and figure 5.9. We remark that in this case we were able to achieve, without running out of memory during synthesis, a queue length of 10 SSTs.



(a) Throughput (expressed in MFLOPS)



(b) Power Efficiency (expressed in MFLOPS/W)

Figure 5.8: Performance measurement of seidel-2D without pipelining enabled within the SST.

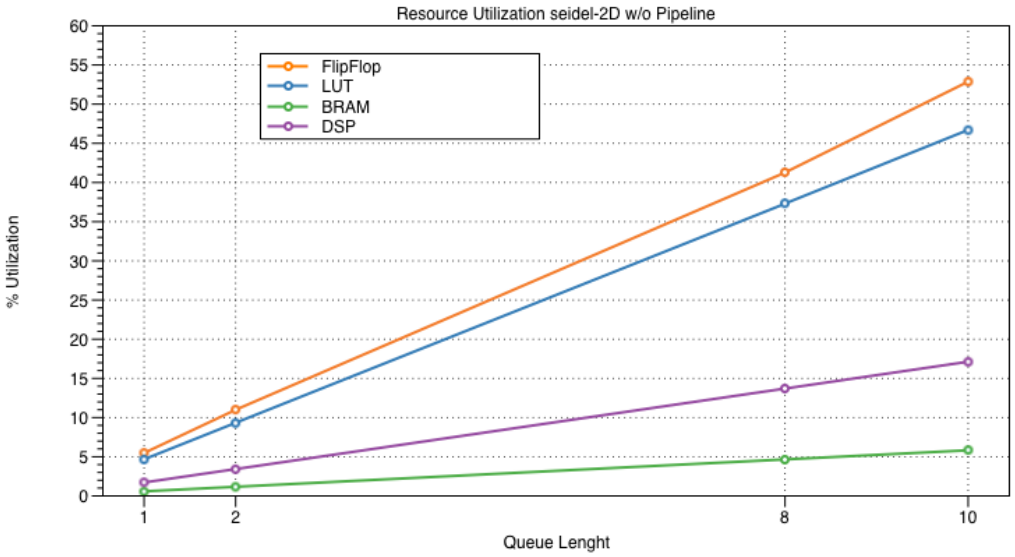


Figure 5.9: Resource usage of the accelerator for the seidel-2D benchmark.

As for the previous two benchmarks, table 5.3 summarize all the results. Due to the presence of spatial dependencies enforce a sequential execution, no parallel CPU implementation could be tested.

Also in this case, the naive implementation could not be implemented, as it exceeded the total available on-chip memory.

Table 5.3: seidel-2D

	Throughput	**Power Efficiency**	**LUT**	**FF**	**DSP**	**BRAM**
	(GFLOPS)	(GFLOPS/W)	(%)	(%)	(%)	(%)
CPU Sequential	0.777	0.049	-	-	-	-
Naive	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
1 SST w/o Pipeline	0.053	0.018	4.670	5.477	1.714	0.583
2 SSTs w/o Pipeline	0.106	0.033	9.331	10.994	3.429	1.165
8 SSTs w/o Pipeline	0.421	0.098	37.311	41.276	13.714	4.660
10 SSTs w/o Pipeline	0.525	0.113	46.715	52.877	17.143	5.825

6

Conclusions and Future Work

This final Chapter provides the thesis conclusions, in Section 6.1, and the envisioned future work, in Section 6.2.

6.1 Conclusions

In this work, we proposed an hardware accelerator to target Iterative Stencil Loops (ISLs), consisting of a queue of microarchitectures demanded to perform a single ISL time-step, the Streaming Stencil Time-step (SST). The power of this accelerator lies in the fact that an SST is able to efficiently exploit the available resources realizing an optimal Full Buffering (FB), but also that the queuing ensure a linear increase in throughput with the increase of the queue length, *i.e.* the number of SSTs within the queue, without increasing the bandwidth demand, which is indeed constant regardless of the queue length. We also proposed a design automation flow to derive the accelerator automatically from the original source code, employing the Polyhedral Model (PM) in combination with the High Level Synthesis (HLS). Experimental results clearly show that the efficient usage of the on-chip memory resources realized by an SST allows to deal with problem sizes that would otherwise be untreatable with a direct synthesis of the original code via HLS, as well as that the SSTs queuing technique ensure a pseudo-linear increase in throughput obtained with constant bandwidth requirements. Also, the comparison made show that the proposed accelerator has the potential to out-

perform all the state of the art solutions thanks to its inherent scalability, but also that the delivered power efficiency rivals the currently available top power efficient systems, and is expected to grow linearly with the increase of the number of SSTs within the queue.

6.2 Future Work

We envision a number of further developments of the proposed work. The first is obviously to implement the automatic framework that realize the proposed design automation flow, following the algorithms and the directives described in chapter 4.

Another important future work is the validation of the accelerator under a multi-FPGA environment, which should however be carried out flawlessly thanks to the inherent scalability of the queuing technique, but also the test of the *queue looping*, that could not be done due to the absence of enough resources as well as the systematic memory overrun issue that limited our synthesis capabilities.

Furthermore, there are two important enhancement that we foresee to unleash the real power of the proposed accelerator: the first is the study and development of techniques to boost the HLS, such as for example an *ASAP scheduling* of the operations inside the computation modules, which are however orthogonal to this work; the second enhancement is to find a way to overcome the bandwidth issue explained in section 5.3.

Another important future work is to better formalize the performance model able to provide a performance estimation given the characteristics of the accelerator, *i.e.* the SST performance and the throughput increase given by the SSTs queuing.

Also, another further development would be the extension of the methodology to deal with periodic boundary conditions, and we suggest that a proper restructuring of the input stream should be enough.

Finally, although we target only the ISLs domain with the proposed acceler-

ator, we envision a direct link with the entire category of Static Affine Nested Loop Programs (SANLPs) thanks to the employment of the PM, therefore a very important possible future work could certainly be the extension of the proposed methodology to the entire class of SANLPs.

Bibliography

[1] Candl (Chunky Analyzer for Dependencies in Loops).
http://icps.u-strasbg.fr/people/bastoul/public_html/development/candl/.

[2] Clan (Chunky Loop Analyzer).
http://icps.u-strasbg.fr/people/bastoul/public_html/development/clan/.

[3] CLooG (Chunky Loop Generator).
http://www.cloog.org/.

[4] FASTER.
http://www.fp7-faster.eu/.

[5] Green500 List, November 2014.
http://www.green500.org/lists/green201411.

[6] Maxeler MaxCompiler.
http://www.maxeler.com/products/software/maxcompiler/.

[7] PoCC (Polyhedral Compiler Collection).
http://www.cs.ucla.edu/~pouchet/software/pocc/.

[8] PolyBench/C.
http://www.cs.ucla.edu/~pouchet/software/polybench/.

[9] The Human Brain Project.
https://www.humanbrainproject.eu/.

[10] Top500 List, November 2014.
http://www.top500.org/lists/2014/11/.

[11] Vivado Design Suite.
http://www.xilinx.com/products/design-tools/vivado.html.

[12] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012.

[13] A.V. Aho. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

[14] 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.

[15] Corinne Ancourt and François Irigoin. Scanning Polyhedra with DO Loops. *SIGPLAN Not.*, 26(7):39–50, April 1991.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Cédric Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis.

[21] 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.

[22] 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.

[23] 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.

[24] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The Polyhedral Model is More Widely Applicable Than You Think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] R. A. Bergamaschi and W. W. Rosenstiel. An Overview of Today's High-Level Synthesis Tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[26] Marsha J. Berger and Joseph E. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. Technical report, Stanford, CA, USA, 1983.

[27] 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.

[28] A.J. Bernstein. Analysis of Programs for Parallel Processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757–763, Oct 1966.

[29] 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.

[30] 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.

[31] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[32] 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.

[33] A. Chambolle. An algorithm for total variation minimization and applications. *Journal of Mathematical Imaging and Vision*, 20:89–97, 2004.

[34] Tatenda M. Chipeperekwa. Caracal: unrolling memory bound stencils. Technical report, - San Diego, La Jolla CA, USA, 2013.

[35] 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.

[36] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[37] 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.

[38] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[39] Hui-Min Cui, Lei Wang, Dong-Rui Fan, and Xiao-Bing Feng. Landing Stencil Code on Godson-T. *J. Comput. Sci. Technol.*, 25(4):886–894, July 2010.

[40] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[41] 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.

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[43] 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.

[44] 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.

[45] 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.

[46] Paul Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[47] Paul Feautrier. Dataflow Analysis of Scalar and Array References. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[48] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *Int. J. Parallel Program.*, 21(5):313–348, October 1992.

[49] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[50] Dietmar Fey. *Grid-Computing*. Springer, 2010.

[51] B. Fischer and J. Modersitzki. Fast Inversion of Matrices Arising in Image Processing. *Numerical Algorithms*, 22:1–11, 1999.

[52] 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.

[53] Matteo Frigo and Volker Strumpen. The Memory Behavior of Cache Oblivious Stencil Computations. *The Journal of Supercomputing*, 39(2):93–112, 2007.

[54] 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.

[55] Heiner Giefers, Christian Plessl, and Jens Förstner. Accelerating Finite Difference Time Domain Simulations with Reconfigurable Dataflow Computers. *SIGARCH Comput. Archit. News*, 41(5):65–70, June 2014.

[56] 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.

[57] Robert M. Haralick and Linda G. Shapiro. *Computer and Robot Vision*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1992.

[58] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[59] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A Stencil Compiler for Short-vector SIMD Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[60] 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.

[61] Pao-Ann Hsiung, Marco D Santambrogio, and Chun-Hsian Huang. *Reconfigurable System Design and Verification*. CRC Press, 2009.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] Jens Krueger, David Donofrio, John Shalf, Marghoob Mohiyuddin, Samuel Williams, Leonid Oliker, and Franz-Josef Pfreund. Hardware/Software Co-design for Energy-efficient Seismic Modeling. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 73:1–73:12, New York, NY, USA, 2011. ACM.

[68] 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.

[69] 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.

[70] Peng Li, Louis-Noël Pouchet, and Jason Cong. Throughput optimization for high-level synthesis using resource constraints. In *IMPACT*, 2014.

[71] 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.

[72] 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.

[73] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[79] 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.

[80] L.R. Pettey and The University of Texas at Austin. Chemistry. *Quantum Dynamics on Adaptive Grids: The Moving Boundary Truncation Method*. University of Texas at Austin, 2008.

[81] Louis-Noël Pouchet. *Interative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, Orsay, France, January 2010.

[82] Louis-Noël Pouchet. Polyhedral Compilation Foundations. Ohio State University. Course Lecture, 888.11, 2010.

[83] 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.

[84] 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.

[85] 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.

[86] Harald Prokop. Cache-Oblivious Algorithms, 1999.

[87] 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.

[88] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of Efficient Nested Loops from Polyhedra. *Int. J. Parallel Program.*, 28(5):469–498, October 2000.

[89] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1:95–113, 1989.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] A. Schafer and D. Fey. High Performance Stencil Code Algorithms for GPGPUs. In *International Conference on Computational Science*, ICCS '11, pages 1–10, 2011.

[99] Michael Schmidt, Marc Reichenbach, and Dietmar Fey. A Generic VHDL Template for 2D Stencil Code Applications on FPGAs. In *Proceedings of the 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, ISORCW '12, pages 180–187, Washington, DC, USA, 2012. IEEE Computer Society.

[100] 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.

[101] 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.

[102] L.G. Shapiro and G.C. Stockman. *Computer Vision*. Prentice Hall, 2001.

[103] 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.

[104] Scott Sirowy and Alessandro Forin. Where's the Beef? Why FPGAs Are So Fast. Technical Report MSR-TR-2008-130, Microsoft Research, September 2008.

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

[110] Jan Treibig, Gerhard Wellein, and Georg Hager. Efficient Multicore-Aware Parallelization Strategies for Iterative Stencil Computations. *CoRR*, abs/1004.1741, 2010.

[111] 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.

[112] 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.

[113] Sven Verdoolaege. Polyhedral Process Networks. In *Handbook of Signal Processing Systems*, pages 1335–1375. 2013.

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] R. Wester and J. Kuper. Deriving Stencil Hardware Accelerators from a Single Higher-Order Function. In *Communicating Process Architectures 2014*, CPA '14, 2014.

[119] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, October 1991.

[120] 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.

[121] 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.

[122] David G Wonnacott and Michelle Mills Strout. On the Scalability of Loop Tiling Techniques. *IMPACT 2013*, page 3, 2013.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

April 8, 2015

Document typeset with LaTeX