POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Explicitly Isolating Data and Computation in High Level Synthesis: the Role of Polyhedral Framework

Relatore: Prof. Marco Domenico SANTAMBROGIO

Correlatore: Dott. Ing. Riccardo CATTANEO

Tesi di Laurea di:

Gabriele Pallotta

Matricola n. 755308

Anno Accademico 2013–2014

This page has intentionally been left blank

We are the hammer! The polyhedral hammer!

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AST	Abstract Syntax Tree
API	Application Programming Interface
ASIC	Application-specific integrated circuit
CANDL	Chunky ANalyzer for Dependencies in Loops
CLAN	Chunky Loop ANalyzer
CLAY	Chunky Loop Alteration wizardrY
CLOOG	Chunky LOOp Generator
CPU	Central Processing Unit
DA	Dependence Analysis
DMA	Direct Memory Access
FIFO	First In First Out
FLOPS	Floating Point Operation Per Second
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
HLS	High Level Synthesis
HDL	Hardware Description Language
ID	Iteration Domain

LeTSeE	LEgal Transformation SpacE Explorator
LLVM	Low-Level Virtual Machine
MP-SoC	Multi Processor-System on Chip
PA	Polyhedral Analysis
PM	Polyhedral Model
PPN	Polyhedral Process Network
PPU	Physics Processing Unit
PoCC	Polyhedral Compiler Collection
PRDG	Polyhedral Reduced Dependency Graph
RAR	Read After Read
RAW	Read After Write
RTL	Register-Transfer Level
SAC	Single Assigned Code
SANLP	Static Affine Nested Loop Program
SCM	Sequential Communication Media
SCoP	Static Control Parts
SOLOMON	Simultaneous Operation Linked Ordinal MOdular Network
SLD	System Level Design
VHDL	VHSIC Hardware Description Language
WAR	Write After Read
WAW	Write After Write
YAML	YAML Ain't Markup Language

Summary

The increased computational power required by modern large-scale computing system is pushing the adoption of heterogeneous components into mainstream. While Graphics Processing Units are frequently adopted as core computational elements, FPGA based heterogeneous systems are being investigated and adopted due to their claimed superiority in power efficiency. However, the lack of proper approaches and methodologies to systematically push the performance of such devices are among the principal factors limiting the adoption of these devices into mainstream. In this paper, I investigate the adoption of Polyhedral Analysis to extract data level parallelism from sequential code, defining a methodology for High Level Synthesis aimed at FPGA based system. I show how our approach systematically produces speedups proportional to the amount of data level parallelism available in the input programs.

Sommario

La sempre più elevata richiesta di capacità computazionali richieste dai moderni sistemi di elaborazione su larga scala sta rendendo sempre attuale e pressante l'esigenza di introdurre soluzioni eterogenee. Se da un lato è ormai consolidato l'uso di GPU come elementi fondamentali del calcolo numerico, dall'altro vi è un sempre più crescente interesse verso sistemi basati su FPGA, e ciò è dovuto soprattuto alla loro intrinseca efficenza energetica. Uno dei principali limiti che ne stanno limitando un'adozione ancora più massiccia, è la mancanza di approcci teorici, metodologici e sistematici. Nel presente lavoro rivolgerò grande attenzione all'adozione delle Analisi Poliedrale allo scopo di estrarre il parallelismo a livello dati dal codice sequenziale costituente gli algoritmi di calcolo presi in esame. Verrà evidenziata una metodologia per la Sintesi ad Alto Livello rivolta ai sistemi basati su FPGA. Mostrerò come l'approccio proposto produca sistematicamente notevoli aumenti di velocità di calcolo, aumenti proporzionali al livello di parallelismo delle strutture dati ricavabile dai programmi di calcolo forniti in ingresso.

1

Introduction

"We [the Moderns] are like dwarves perched on the shoulders of giants [the Ancients], and thus we are able to see more and farther than the latter. And this is not at all because of the acuteness of our sight or the stature of our body, but because we are carried aloft and elevated by the magnitude of the giants."

– Bernardo of Chartres

In this Chapter I introduce technologies involved in this work and what power efficiency means in the context of computing systems. Also, I describe the rationale behind my work and the goal I am trying to achieve considering the current technological trends in hardware development.

1.1 Background

In the last two centuries, industry and technology grew on two assumptions about energy: infinite availability of raw materials, and a cost of energy that can be neglected. In the past, energy costs were one or more order of magnitude lower than technical costs.

In recent decades, many energetic crisis have muted the entire landscape. As energy related costs are becoming more and more significant, and overall resources costs arise as they are become scarcer and scarcer, it is mandatory to extract the most from every resource spent. Governments of the major industrial countries, in E.U. and U.S., created specific entities to control and improve the energy usage of they respective countries. Wasting energy isn't affordable anymore, and science has to put forth, too.

In this section, I will analyze some of the key aspects that lead inefficient usage of electricity supplies in high performance computing and the major approaches to address this problem.

In particular I discuss:

- On Power Utilization
- Power Efficiency
- Programmability
- Trend Analysis in High Performance Computing

In the following pages I will introduce some basic terms usually used in electronics and physics to describe electrical and thermal behavior os silicon circuits in order to explain mutual relation between current, power consumption and frequency.

1.1.1 On Power Utilization

The most simplified but overall valid model to electrically power consumption in digital circuits has two terms:

- static
$$P_{static} = I_{static} * V_{dd} \tag{1.1}$$
- dynamic
$$P_{dynamic} = P_{cap} + P_{transient} \propto (C_L + C) * V_{dd}^2 * f \tag{1.2}$$
- total power can be obtained adding static and dynamic power:
$$P_{total} = P_{static} + P_{dynamic} \tag{1.3}$$

From the equation we note that dynamic power follows a quadratic relation with voltage and linear relation with frequency. During actual computation the dominant term used to be the dynamic part. However, new developments are changing the ratio between static and dynamic power, leading to the static part to be comparable to the dynamic. In fact, reduction of feature size, capacitance's will greatly decrease so dynamic power will become comparable to static power and we must take into account both terms.

In order to increase computational power, from an electronic point of view, we mainly have two ways:

- increase frequency
- increase the number of transistors

If we increase frequency, the gain in computational power is linear. Unfortunately, since transistors have physical limits, we cannot reduce the voltage below a given threshold voltage. So, when we aim for higher frequency we also need greater voltage to allows correct charging of capacitors. As we previously stated, an increment in voltage yields a quadratic increase in power consumption.

If we increase the number of transistors we can get more computational power lowering frequency, avoiding the dramatic heat dissipation due to voltage increase. On the other hand, more transistors means bigger static power consumption, and allow wider and multiple (potentially parallel) components [1].

In processing units we can identify two different sets of transistors are used:

- Those employed to produce computation
- Those employed to perform non computational tasks such as prefetching data, caching and decoding instructions; i.e. to improve performance of regular but most importantly irregular computation

By allocating transistors to the first set, a processor gains an increment in the computations per Watt ratio. On the other hand, transistors dedicated to the second set lower the same ratio, but allow better software programmability and greatly simplify software designs.

1.1.2 Power Efficiency

In a world where technological development poses more and more challenges, the scientific community is facing the problem os relatively scarcer and scarcer affordable computational resources (with respect to problem size). On the other hand, extreme-scale computing will enable the solution of vastly more accurate predictive models and the analysis of massive quantities of data [2, 3].

In order to solve these kind of problems, industrialized nations are aiming at *Exascale computing* systems. A machine, to be classified as exascale, must be capable of processing at least one exaFLOPS (10^{18}FLOPS). Problems amenable to these machines only includes [4]:

- Efficiency and safety of nuclear energy sector (4th generation, specifically [2])
- Reverse engineering of the human brain
- Dramatically improved regional climate models capable of better predicting changes such as sea level rise, droughts and floods, and severe weather patterns

These are only few problems an exascale machine could solve [5]. However, in order to generate such vastly amount of computational power. With current technological solutions, it is require too much power. The rising need of exascale machines brought DOE, and other scientific organizations to issue the exascale initiative [6]. One of the most important limitations imposed by this challenge is to create said machines with a power budget of roughly 20 MW of power.

Due to these limitations in high performance computing, it is mandatory to refer as performance not as raw computational power, but as how many calculations you can do for every single Watt spent to do so.

1.1.3 Programmability

In CPUs and GPUs a great amount of power is spent on operations that do not produce actual computation. In the past, a lot of work was made to improve performance of irregular or general code from processing units. Since processors were created to perform a lot of different tasks, they needed a method to do so: the basic *fetch-decode-execute cycle* was invented. When performance became an issue, companies upgraded this technology by implementing the *instruction pipeline*, allowing for much shorter critical paths, and then, higher operational frequencies. When they discovered the basic principles of data locality they implemented *caches*, and again when they understood how to predict mutual behavior in conditional statements they introduced the *branch history tables*. As the race for performance continue, engineers struggle to refine and enhance processors with the goal to further improve their throughput. All of the above solutions have been introduced for while (mostly never) scarifying programmability mainly due to market reasons. Eventually, another huge problem arise: *power consumption*.

All of the improvements done so far were developed with throughput in mind. No - or few - considerations were made on power consumption. As transistors feature shrunk and reached their physical miniaturization limits, processors became ultra-dense components unable to be powered on completely, unless melting them was the actual goal. This was the advent of the *Dark Silicon* [7] era, where we not only limit operating frequency but also the amount of parallel computation that it is possible to carry out inside a single chip package.

However, more efficient patterns and architectures exist. Their drawback relates to the programmability model as they relies on many different specialized components that must be programmed accordingly. GPUs belong to this family. Given their parallels nature, a huge effort must be put to program them in order to achieve the best performance available.

As we see later, GPUs are not the only family. Actually exists another architecture that is able to deliver a huge amount of throughput with very low power usage, at the cost of programmability, as they are the most dissolute device to program (with regards to CPUs, and even GPUs).

1.1.4 Trend Analysis in High Performance Computing

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

Modern datacenters requires big investments in electricity supplies not only to supply electronic equipments but to cool them down, too. Additionally electricity power drawn by the datacenter is hitting limits imposed by utilities companies in most places, as well. As power efficiency not only reduces costs in electronics but allow bigger savings in conditioning systems, governments are putting a lot of resources to incentive the researches of new techniques to reduces the energy consumed by those systems [8].

The most relevant trend harnessing this problem is *heterogeneous computing*. Heterogeneous computing refers to systems that use more than one kind of processing units. These are systems that gain performance not just by assembling more components of the same type, but by adding customized processing units, usually incorporating specialized processing capabilities to handle particular tasks. Since these components are suited only for a specific task a far lower number of transistors are usually required in order to process them. Specialized components usually work at lower frequency, too, reducing, as stated in 1.1.2, the overall power consumption of the system. For these reasons, mathematical co-processors where introduced in late 80's. This was the first example of heterogenous system. A more relevant and modern example in this direction is the introduction of GPUs, that were initially used to accelerate the compute-intensive work of texture mapping and polygon rendering. Afterwards, units were added to accelerate geometric calculations such as the rotation and translation of vertices into different coordinate systems.

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

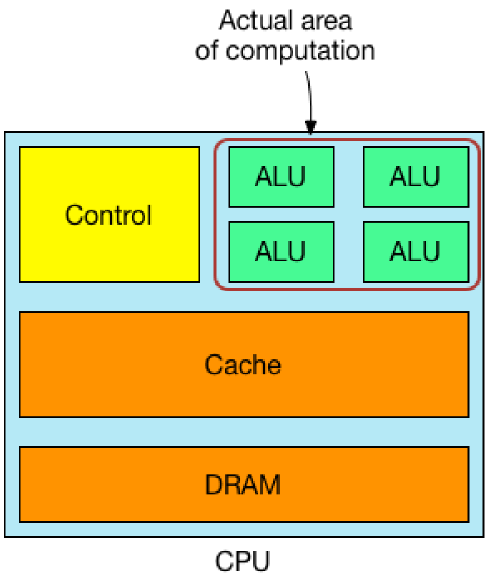


Figure 1.1: CPU horsepower

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

Another important direction in heterogeneity is the introduction of physics chips: they offload physics calculations from the CPU, and are performed on dedicated hardware circuit (for example PhysX[9], is a proprietary realtime physics engine middleware SDK, born from an hardware solution by Ageia, that called it Physics Processing Unit (PPU)).

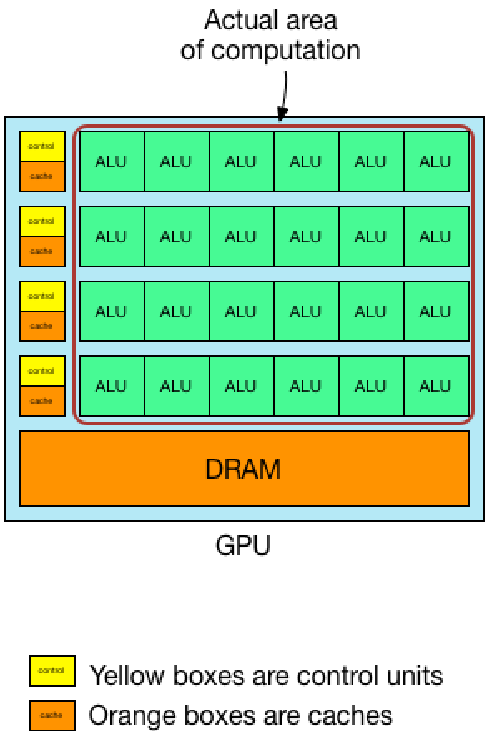


Figure 1.2: GPU horsepower

Phi cores [10] are based on the same idea of the multicore architecture, but relying on more, less complex micro processors. The basic idea that led to the creations of such component is that these cores can retain many of the existing programming models that most developers are familiar with.

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

Another technology on the rise that shows an interesting set of features is basically the FPGA. FPGAs are component designed with a completely different goal in mind. FPGAs have no mathematical or logic components per se, but have to be arranged in order to implement those functionalities. Another aspect of FPGAs is *Dark silicon*: In FPGAs only transistors in configured circuits are powered on. In this way it is possible to achieve increased power efficiency.

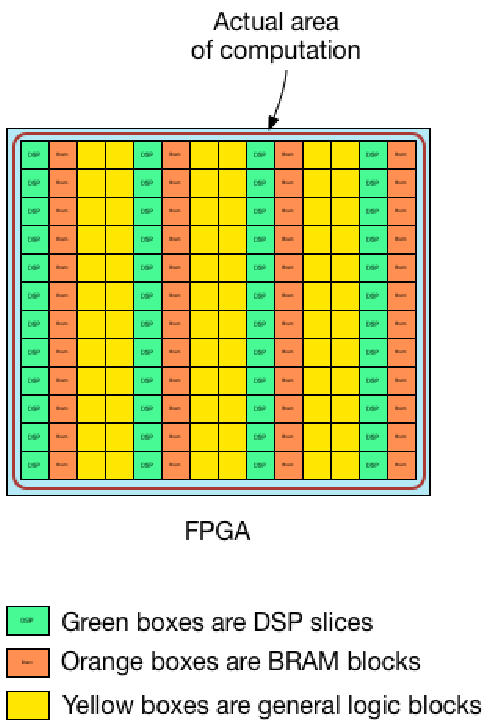

Actual area of computation
DSP
Bram
FPGA
Green boxes are DSP slices
Orange boxes are BRAM blocks
Yellow boxes are general logic blocks

Figure 1.3: FPGA horsepower

However, FPGAs must be configured in order to obtain power efficient processing units out of them. Such process is very complex as it involves hardware design. This needs to be repeated for every problem at hand, resulting in a very time consuming process. However, the creation of custom architectures fitted on the algorithm will results in huge energy savings, incrementing the power efficiency of the system. Not all the problems can take advantage of this approach, but many can be efficiently implemented.

Programmability is an issue as Field Programmable Gate Array (FPGA)s are programmed in a very different manner than CPUs and GPUs. Current research - both industrial and academic - is focusing on improving the experience of software developers as they should only concentrate on software algorithms leaving a sophisticated toolchain the burden to implement it as dedicated circuits (for example, Xilinx[11] with SDAccel SDK [12]).

As the FPGAs approach is radically different to the CPUs and GPUs, the next Section is dedicated to how computation is described for this devices.

1.2 Hardware Acceleration

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

1.2.1 What is Hardware Acceleration

Hardware acceleration is a technique that consists in implementing some, or all, parts of an algorithm via dedicated hardware circuits. Said circuits produce the same results as their software counterparts [13, 14, 15, 16]. Traditionally, the hardware designer was in charge to creating the circuits by hand. So, he had to have a great understanding of hardware components and how they could be connected in order to achieve the corresponding algorithmic operation. The entire workflow is very time consuming, involved and error prone but nonetheless required when the goal is to achieve the best performance available. This workflow will make extensive use of Hardware Description Language (HDL), which are difficult to understand and manage for most software designers who usually are the ones in charge of coding algorithms. To make a comparison between hardware and software development, HDL based development resembles the use of Assembly to optimize custom routines in C/C++ development. Since HDLs were developed to describe hardware circuits, they are characterized by a low level of abstraction. Thus, hardware designers must take into account every single detail such as signals, state machines and their behavior over time. Also, debugging at this level is very complex and an hardware and electronic knowledge is required to understand waveforms, timing constraints and their impact in the final design.

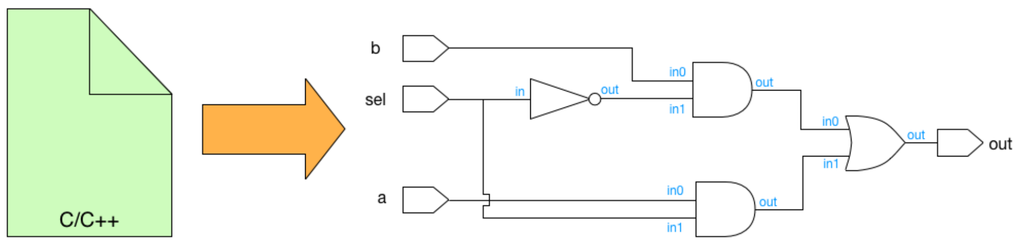


Figure 1.4:
Hardware implementation
of IF statement

1.2.2 Why to employ Hardware Acceleration

As explained before in Section 1.1 and subsection 1.2.1, the implementation of a dedicated hardware component has the major benefit of speeding up portions of an application. In fact, it is usually true that there is no CPU program that can run as fast as a dedicated circuit given that the latter is comparable in terms of technology and frequency to the former. This is due to the overhead needed to maintain the CPUs as a general processor as possible (i.e. in order to compute anything the software developer can think of).

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

As programming GPUs is a very complex task, major vendors put a lot of efforts into introducing a set of a APIs and libraries to make the process easier. Notable examples are Nvidia Cuda [17] and AMD Mantle [18] frameworks. Moreover they show relatively low power efficiency with regards to FPGA [19, 20, 21] on most workloads, for the aforementioned reasons.

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

To summarize, GPUs can be considered suboptimal for high consumption, low power efficiency.

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

1.2.3 High Level Synthesis

While HLS tools have been heavily studied in the past, only in the recent years we have seen effective industrial tools available in the market. Current research is focusing on efficiently converting numeric or image processing algorithms written in behavioral languages directly into hardware implementations in order to achieve better performance and lower consumption while highering the layer of abstraction in order to gain in designer programmability. This has been possible in the recent years because High Level Synthesis (HLS) tools have become powerful and flexible enough to allow relatively easy and fast synthesis of hardware circuits. Previously, hardware development required plenty of specific knowledge in order to develop a fully working accelerator. High Level Synthesis (HLS) tools impose less requirements on designers and dramatically speeding up the development of a working system. However, without proper care, this comes at the cost of introducing large overheads and slow-downs compared to manually designed implementations. This is due to the lack of knowledge that High Level Synthesis (HLS) tools have in order to do optimizations on the resulting components. In order to cope with these limitations, High Level Synthesis (HLS) tools has special directives that can be used to optimize the resulting components, with the only downside that these directives need to be specified by the designer and are not derived automatically.

1.2.4 Optimize High Level Synthesis

HLS has a lot of directives allowing to generate different components [22, 23], each with its own specific performance profile. Some of them are useful to increment the throughput, other to minimize the area and others again are explicitly used for lowering the power consumption. For example the *dataflow* directive can be used to parallelize function calls and/or nested loops creating different blocks of circuits inside a single core, each capable to run concurrently. This directive also looks at and preserves the dependences in the code to maintain the correctness of the output. Another useful directive is the *pipeline* directive. This directive tells the HLS tools to use more resources in order to create a pipeline inside the core, or in case this directive is used with the *dataflow* directive, to create a pipelined block inside the core. Other directives such as, *array map*, *array reshape* or *array partition* serve the purpose to optimize the number of BRAMs used inside the FPGA. Another useful directive is *unroll*. This feature can partially or completely unroll a loop in order to run in parallel all its iterations of a loop body. As mentioned in 1.2.4, these directives have to be explicited by the hardware designer.

1.2.5 Input languages to HLS tools

Current hardware circuits can be generated in very different ways.

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

As previously stated in 1.2.2, HLS tools are getting more and more powerful, closing the gap between automatic and manual implementation; plus, they allow the creation of RTL from high level language such as C/C++, or with the newer OpenCL C [24].

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

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

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

- most syntax analyzers and compilers are written for C/C++ so its easy to get robust tools to further enhance code deriving from them
- It's easy to simply port algorithms from a platform to another and to HW, too, as C is well defined and standardized
- Support a familiar "hardware level of abstraction", providing a link between high-level source code and low-level implementation [25]

It's a matter of fact that there is no specific reason we cannot start from another language (say Java, Haskell or other languages) but C/C++ is the de-facto standard in industrial development. The vast majority of software developers write complex algorithms relying on C/C++ features, so it would be very unproductive to force them to learn another language and revolutionize all their fine-tuned coding practices.

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

- No aliasing (i.e. Fortran)
- All parameter passing is done by value (we solve from language itself some synchronization issues, i.e. Haskhell, but we do not resolve communication issues)
- Passing arguments by value will waste memory very quickly and so we need to rethink the algorithm in a more efficient way

Note that other languages can also use other means to get parallel/optimized computation: for example, in Haskhell you get for free fast lightweight threads, parallel sparks and futures, software transactional memory, core affinity control and so on. However, such features mostly cannot be ported to HDL (even if there are project like [26, 27, 28, 29, 30, 31, 32, 33] that aim for it).

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

1.3 Application Domain

The class of problem I am targeting are all the *scientific workloads*. In fact, these algorithms can be easily written as:

- Static
- Pure
- Affine

imperatives codes.

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

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

1.3.1 Staticness

Given a C code, we can define it *static* if:

- All loop bounds are known at compile time
- There are no data dependenct conditional statements

Pseudocode 1 Example of a static code

```
define M 10
define N 10

for i=1 to N do
  for j=i to M do
    if j <= 2 then
      b[j] = Func()
    end if
  end for
end for
```

1.3.2 Affinity

Given a code we can define it *affine* if accesses to arrays happen using indeces, constants or linear combinations of the indeces of the enclosing loops. For example, Code 1 is also affine since data are also accessed linearly using j alone. An example of an affine but not static code is:

Pseudocode 2 Example of an affine non static code

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

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

Note: in line 5 the *if-statement* depends on data value, breaking the second condition for staticness. Each index in code 2 is a linear combination of enclosing indexes and constants.

1.3.3 Pureness

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

Pure functions

A function is pure if:

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

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

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

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

Pseudocode 3 Example of a pure code

```
define M 10
define N 10

func foo()
a[]
for i=1 to N do
    for j=i to M do
        if j <= 2 then
            b[j] = Func(a[i])
        end if
    end for
end for
endfunc
```

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

Pseudocode 4 Example of a non pure code

```
define M 10
define N 10
a[]

func foo()
for i=1 to N do
    for j=i to M do
        if j <= 2 then
            Func(&a[i])
        end if
    end for
end for
endfunc
```

Beauty of pure functions

Pure functions map well on parallel hardware, since it won't be required any global memory and potentially critical bottleneck in most systems. Indeed, implementing global state on hardware will require hardware synchronization mechanism, wasting precious resources and will introduce wait states for all the components that rely on that information. Pure code prevents by design these kind of side effects.

1.4 Long Term Vision

Trends described in 1.1.4, strongly hint to a future where heterogenous systems are the norm. More and more datacenters and supercomputers are relying on heterogeneity to achieve faster and faster computing speed while maintaining the power consumption as low as possible.

Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts, but they allowed to quickly prototype components or build up circuits when ASIC production would be too expensive. Nowadays, thanks to technology advancements, FPGAs can realistically be seen as the next core heterogenous components in (near) future supercomputing. Right now, researchers are porting algorithms on this platform to achieve better throughput at lower power consumption than their GPUs counterpart [34, 35, 36, 37, 38, 39].

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

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

This work elaborates on a novel way in support this trend

The rest of the dissertation is based on this prediction, thus presenting the corresponding state-of-art in high performance computing in Chapter 2. The problem statement follows in Chapter 3. An innovative methodology to extend the discipline is described in Chapter 4. The results are shown in Chapter 5 and final conclusion are drawn in Chapter 6.

2

State of the Art

The state-of-the-art in this field of research has ancient roots. Since the early 60’s , after the advent of first integrated circuits computers, researchers started to think of a way to achieve better performance from this machine. In 1966 A.J. Bernstein explained which general conditions allow parallel processing and the memory organization needed in a multicomputer system in order to achieve it. In that paper [40] he asserted that, even at that time, the idea of processing a program in parallel was not new: in those days was designed a sophisticated machine called "Simultaneous Operation Linked Ordinal MOdular Network (SOLOMON) computer" [41] which could solve problems composed of a number of identical, independent calculations, for example involved in the solution of partial differential equations. In the same paper Bernstein showed that the decision if two tasks can be executed in parallel depends on quality of algorithm and on specific implementation. At that time he inferred that knowing the assumption: "two program blocks are parallel if and only if they produce the same results when performed sequentially or in parallel for all possible sets of input data", is equivalent to solve "the halting problem for an arbitrary Turing Machine T starting with an arbitrary initial tape".

This problem is well known in literature, and is also known to be an *undecidable* problems. Even if, the terms undecidable means that there is no program that can answer if two parts of a program can run in parallel, Bernstein came up with the condition (Bernstein condition’s) that allow us to know if some parts of

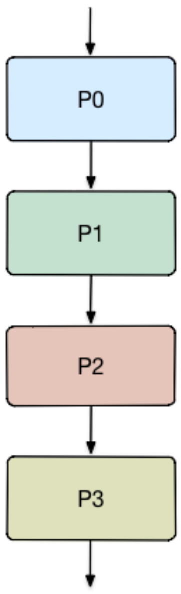


Figure 2.1: Sequential execution of a program

a program can be run at the same time. We can summarize those conditions in the following statement: "There must be no dependency between the parts of a program".

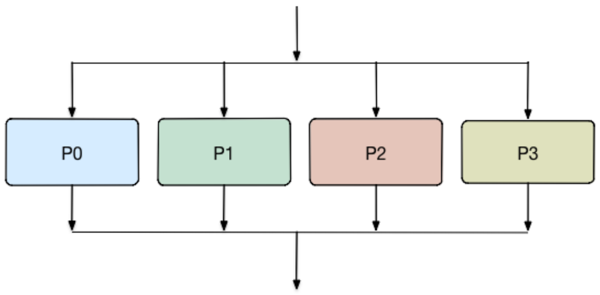


Figure 2.2: Parallel execution of a program

As of what we wrote our work seems pretty useless, but the lack of a generic algorithm it doesn't mean that there is no algorithm. Obviously, it must be a problem specific algorithm that can answer if two parts can run in parallel. So if we can reduce some problem to a known problem which we already know how to parallelize, we can also infer how we can parallelize it. This is exactly what we are aiming using the framework known as Polyhedral Model (PM). The PM can be applied only on algorithm written using static code. So we can argue that any problem written with code of which information are all known at run time can be taken and analyzed automatically using PM. Snatching what part of a problem can be parallelized is of utmost importance in speeding up every application, it is even more important since we want to create an hardware architecture in order to hardware-accelerate algorithms. The main motivation to accelerate algorithms through hardware circuits is the upcoming and foreseeing end of the "Moore's law". Previously, in 1965, Moore had predicted that the number of transistors on a semiconductor (and thus the overall chip performance) would double every two years . Moore also stated that "no physical quantity can continue to change exponentially forever", due to the miniaturization of transistors that would reach its physical limits and it could not further allow to produce faster processors. The law has demonstrated to be correct for many years and it continued to be valid in different ways, by producing more powerful processors, multi-core processor architectures. Multi-core processors consist of processors, usually of the same type, built and integrated into a single chip. In recent years, we saw the rise of heterogeneous architectures, which have dedicated components suited for dedicated tasks, and as the time passed the area on chip becomes more and more used on dedicated components.

Since an algorithm that suits best for hardware acceleration can also be optimized with PM, we want to exploit the latter to generate better hardware.

In the following section we will introduce the PM.

2.1 Polyhedral Model

In Chapter 1 I talked about the new trends in computer technologies. As the number of transistors on a single die started to rise, power consumption ad heat dissipation became more and more complex to manage [7]. Multi-cores architectures have been introduced to mitigate increasing consumption problem keeping the same computational power. These architectures operate at generally lower frequencies, but with multiple cores we can achieve even better performance, rising the Moore's law to a whole new level. With the advent of these true parallel architectures new problems appeared, such as consistency of data between cores, needs to re-think code to get better parallelization, lock and synchronization issues, only to name a fews. To squeeze all the power from parallel hardware architecture we need new skills to perform complex loop nest restructuring in order to write better optimizing and parallelizing tools. The polyhedral model has demonstrated its potential to enhance performance over a variety of targets. In this Chapter we will discuss the theoretical terms needed to understand how polyhedral analysis works and why we need it in order to exploit parallelism from static code.

Let's introduce some definitions:

2.1.1 Polyhedral Model

A polyhedron is set of rational values described by affine inequalities.

Polyhedron

The intersection of a finite set of closed linear half-spaces is called a Polyhedron and is specified by a system of linear equalities and inequalities;

$$P : \left\{ \vec{x} \in Q^n | A\vec{x} \geqslant \vec{b} \right\} \tag{2.1}$$

where A is j x n matrix, $\vec{b}$ is a j-vector and n is the dimension of space that contains the polyhedron. Smallest affine subspace which spans the polyhedron determines the dimension of the polyhedron.

Parameterized Polyhedron

Parameterized Polyhedron $P(\vec{p})$ is described as linear function of p which is an m-vector of parameter;

$$P : \{\vec{x} \in Q^n | A\vec{x} + B\vec{p} \geqslant \vec{c}\} \tag{2.2}$$

where A and B are constant matrixes and $\vec{c}$ is a constant vector. Input program (source code) is usually represented in some internal representation form in compiler's domain. In most conventional compilers this form is the Abstract Syntax Tree (AST). This form allows manipulation and optimization on the code. Polyhedral Model is one this special representation form considered useful for parallelizing codes. The model is applied to affine nested loops in compiler optimizations to efficiently analyze and transform the source code.

Iteration Domain

Set of values of an iteration vector for which a statement is executed. $D(S)$ stands for the Iteration Domain of statement S. An iteration vector $\vec{x}$ of a statement is built from the iterators of surrounding for and while loops of the statement. If a while loop is not mentioned explicitly, a virtual iterator $w : 0 \leqslant w$ is associated with that loop. For Example, if we take the sample of pseudo code below:

Pseudocode 5 Example of an algorithm

```
parameter M 1 10
parameter N 1 10

for k=1 to M do
    S1: y[k] = F1()
end for
for i=1 to N do
    for j=i to M do
        if j <= 2 then
            S2: y[j] = F2()
        end if
        S3: [] = F3(y[j])
    end for
end for
```

We can derive the linear inequalities that describe the geometry of the polyhedron corresponding to the Iteration Domain (ID) of the statement S2:

$$D = P(M,N) = \left\{ (i,j) \in Q^2 | \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} \text{i} \\ \text{j} \end{pmatrix} \geqslant \begin{bmatrix} 1 \\ -N \\ 0 \\ -M \\ -2 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} M \\ N \end{pmatrix} \geqslant \begin{bmatrix} 1 \\ -10 \\ 1 \\ -10 \end{bmatrix} \right\}$$
$$= \left\{ (i,j) \in Q^2 | 1 \leqslant i \leqslant N \wedge i \leqslant j \leqslant M \wedge j \leqslant 2 \wedge 1 \leqslant M \leqslant 10 \wedge 1 \leqslant N \leqslant 10 \right\} \tag{2.3}$$

The following graphical representation can be useful to better understand what is the region of the polyhedron.

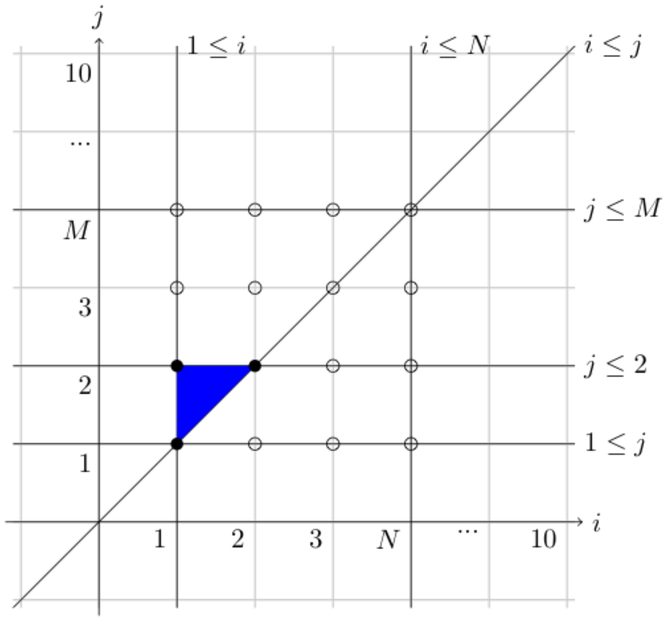


Figure 2.3: Geometrical representation of iteration domain of statement S2 of 5.

2.1.2 Order of Execution

Statements evaluate data in affine nested loops. Evaluation of a statement W on iterator $\vec{x}$ is called an operation and denoted as $\langle W, \vec{x}\rangle$, where $\vec{x} \in D(W)$. Execution order of all operations of all statements is called *the schedule*.

$$\langle W, \vec{x} \rangle \prec \langle R, \vec{y} \rangle \equiv \vec{x}[1...N_{WR}] \ll \vec{y}[1...N_{WR}] \vee (\vec{x}[1...N_{WR}] = \vec{y}[1...N_{WR}] \wedge W \lhd R) \quad (2.4)$$

This equality describe a schedule. If the iterator $\vec{x}$ always precedes iterator $\vec{y}$ or if the iterator $\vec{x}$ is equal to iterator $\vec{y}$ but the statement W precedes the statement R, then operation $\langle W, \vec{x} \rangle$ is evaluated before operation $\langle R, \vec{y} \rangle$.

2.1.3 Definition of Topic Related Terms

In this Subsection I will define all the main terms I will use in the following Sections when discussing about Polyhedral Model (PM) and Polyhedral Analysis (PA).

Static Control Parts

Static Control Parts (SCoP) are a subclass of general loops nests that can be represented in the polyhedral model. A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop and the parameters (constants whose values are unknown at compilation time). The iteration domain of these loops can always be specifed using a set of linear inequalities defining a polyhedron.

Static Affine Nested Loop Program

A Static Affine Nested Loop Program (SANLP) consists of a set of statements and function calls, each possibly enclosed in loops and/or guarded by functions. Loops do not have to be perfectly nested. All lower and upper bounds of the loops, expressions in conditions and array accesses have to be affine functions of the enclosing loop iterators and static parameters. Parameters are symbolic constants: their value should be determined at compile time, no change is allowed during run-time. Data communication between functions must be explicit.

Pseudocode 6 SANLP: An example pseudo code of a SANLP

```
parameter N 10 100

for j=1 to 6*N-3 do
    A[j] = Func1()
end for
for j=1 to N do
    for i=j to 3*j-2 do
        if i+j < 4*N-6 then
            A[i] = Func2(A[2*i-1], A[2*i+1])
        end if
        Func3(A[i])
    end for
end for
```

Polyhedral Reduced Dependency Graph

A graph where nodes represent computation and edges represent communication. Nodes communicate point-to-point via unique multi-dimensional arrays which suit original data dependencies.

Polyhedral Process Network

Target Polyhedral Process Networks (PPN) [42, 39] is a special case of Kahn Process Networks (KPN) model of computation. A PPN consists of concurrent autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels using a blocking read/write on an empty/full FIFO as synchronization mechanism. Everything about the execution of a PPN is known at compile-time. Automatic synthesis can perform calculation of buffer sizes which guarantee a deadlock-free execution.

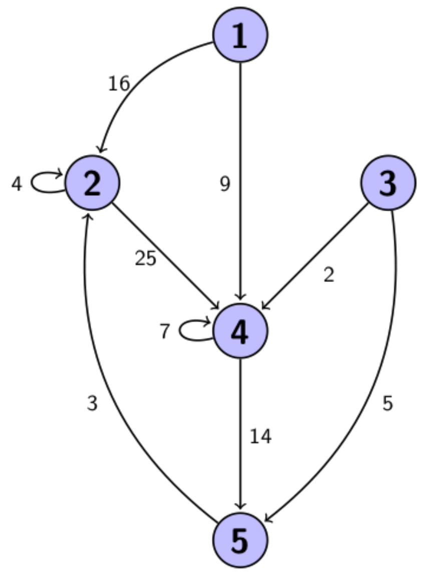


Figure 2.4:
Sample of a PPN
the values on the edge are only to show their weight.

2.1.4 Polyhedral Analysis

In most cases, a completely sequential execution can be parallelized without compromising the correctness of the execution. The order of the instructions can be rearranged without changing the program functionality and respecting the ordering constraints. If we can rearrange statements, we implicitly get a big canche to execute them in parallel. Ordering constraints are dictated by the data dependency relations existing in the sequential program. Therefore, the first main step of the parallelization is to perform data dependency analysis. The analysis helps to extract the dependent statements and presents an initial program in a way where data dependencies are made explicit. Thus, the initial program is translated into the Single Assigned Code (SAC) form or its analogous form called Polyhedral Reduced Dependency Graph (PRDG) which is a compact mathematical representation of the dependency relations in terms of polyhedra. In PRDG the nodes represent statements of the initial program and the edges represent data dependencies. The PRDG model exploits multi-dimensional arrays for data communication,while he target model PPN, requires First In First Out (FIFO) channels as communication medium. Therefore, another step is needed to convert multi-dimensional memory access scheme into managed dataflow over FIFO channels. This is called Linearization.

2.1.5 Dependence Analysis

In compiler theory, dependence analysis produces execution-order constraints between statements. We say statement S_2 depends on S_1 if S_1 must be executed before S_2. Is it possible to individuate two major classes of dependencies:

- control dependencies
- data dependencies

Dependence analysis is important because determines whether or not it is safe to reorder or parallelize statements.

Control Dependencies

An instruction is control dependent on a preceeding instruction if the effect of the latter determines whether the former should be executed or not.

Pseudocode 7 RAR: An example pseudo code of a RAR dependency

```
1: if A == B then
2:     A = A + B
3: end if
4: B = A + B
```

In this example instruction 2 is control dependent on instruction 1. Intuitively we can give the two conditions of control dependance between two statement S_1 and S_2:

- S_1 could be possibly be executed before S_2
- the outcome of S_1 will decide whether S_2 will be executed

Defining the dominance and post-dominance concept we can give a simpler definition of control dependency:

Dominance: In control flow graphs, a node *d* dominates a node *n* if every path from the entry node to *n* must go through *d*.

Post-Dominance: Analogous to the definition of dominance above, a node *z* is said to post-dominate a node *n* if all paths to the exit node of the graph starting at *n* must go through *z*.

Given the above definitions of dominance and post-dominance, we can say that a statement S_2 is said to be control dependent on another statement S_1 if and only if:

- S_2 post-dominates all S_i
- S_2 does not post-dominate S_1

Where an S_i is a statements after S_1 but before S_2, and must be true for all S_i.

Data Dependencies

A data dependency in computer science is a situation in which a program statement (instruction) refers to the data of a preceding statement. In compiler theory, the technique used to discover data dependencies among statements (or instructions) is called dependence analysis.

There are four types of data dependencies:

- input dependency, called Read After Read (RAR)
- flow dependency, called Read After Write (RAW)
- anti-dependency, called Write After Read (WAR)
- output dependency, called Write After Write (WAW)

Only three of them have consequence on the code. Since RAR dependencies only read data and are not harmful are not considered as hazards. Here are included them only for the sake of completion.

Read After Read (RAR) We have an input dependency when we have:

Pseudocode 8 RAR: An example pseudo code of a RAR dependency

```
1: B = A[i]
2: C = A[i]
```

Since this is not a real dependency, because no data is modified, there are no problem if we fall in this case.

Read After Write (RAW) We have flow dependency if an instruction depends on the result of a previous instruction:

Pseudocode 9 RAW: An example pseudo code of a RAW dependency

```
1: A = 3
2: B = A
3: C = B
```

This dependency are often called true dependency. In fact, this dependences aren't avoidable. In the simple example above is not possible to run in parallel the three instruction since each instruction depends on the another previous istruction, hence a level instruction parallelism is not an option.

Write After Read (WAR) We have an anti dependency when we have:

Pseudocode 10 WAR: An example pseudo code of a WAR dependency

```
1: B = 3
2: A = B + 1
3: B = 7
```

An anti-dependency is an example of a name dependency. That is, renaming variables we could remove the dependency:

Here we have removed the WAR dependency but we have introduced a flow dependency between statement 2 and 3.

Pseudocode 11 WAR: An example pseudo code of a WAR dependency simplification

```
1: B = 3
2: B2 = B
3: A = B2 + 1
4: B = 7
```

Write After Write (WAW) We have an output dependency when we have:

Pseudocode 12 WAW: An example pseudo code of a WAW dependency

```
1: B = 3
2: A = B + 1
3: B = 7
```

As with anti-dependencies, output dependencies are name dependencies. That is, they may be removed through renaming of variables, as in the following modification of the previous example:

Pseudocode 13 WAW: An example pseudo code of a WAW dependency simplification

```
1: B2 = 3
2: A = B2 + 1
3: B = 7
```

In real-word mathematical algorithms we have much complex, code, not limited to simple instructions. We usually have multiple loops, with variable nested deep and complex conditions and dependencies. So we need to take analyze and understand mutual dependencies between different variables in different level of nesting. At this I must introduce the definition of the Loop-Carried Dependencies [43]

Loop-Carried Dependencies Since loops are a way to run the same instruction with different data in an automatic way, the question we will want to answer is: "Can two different iterations execute at the same time, or is there a data dependency between them?"

Consider the following loop:

Pseudocode 14 Loop example: An example pseudo code of a completely parallelizable loop

```
1: for i=1 to N do
2:     A[i] = A[i] + B[i]
3: end for
```

Looking at this loop, to answer the question above one should first answer: "Is it possible for any two values of I and J, to calculate the value of A[I] and A[J] at the same time?"
The answer will be more obvious if we manually unroll some iteration of the loop:

Pseudocode 15 Loop example: An example pseudo code of a completely parallelizable loop unrolled

```
1: A[i] = A[i] + B[i]
2: A[i+1] = A[i+1] + B[i+1]
3: A[i+2] = A[i+2] + B[i+2]
```

Looking at the unrolled loop is trivial to understand that this loop is completely parallelizable. Since no statement depend on another, you don't need the results of the first to determine the second. In fact, mixing up the order of the calculations won't change the results in the least. Relaxing the serial order imposed on these calculations makes it possible to execute this loop very quickly on parallel hardware.
Obviously this is an ideal case, in which the are no dependencies between statement. Loop-Carried Dependencies aren't different kind of dependencies than the ones expressed before, hence we can have the same three main types of hazard: flow, anti and output dependencies. The only differences is that are between different iteration of the same statement.

Loop-Carried Read After Write (RAW) Dependencies To understand hazard these dependencies carry, look at the following example:

Pseudocode 16 Loop-Carried Read After Write (RAW) Dependencies example

```
for i=1 to N do
    A[i] = A[i-1] + B[i]
end for
```

This loop can look similar to the previous example, but one of the subscripts is changed. Again, it's useful to manually unroll the loop and look at several iterations together:

Pseudocode 17 Loop example: An example pseudo code of a completely parallelizable loop unrolled

```
A[i] = A[i-1] + B[i]
A[i+1] = A[i] + B[i+1]
A[i+2] = A[i+1] + B[i+2]
```

In this case, there is a dependency issue. The value of the third statement depends on the second one, and the second one depends on the first. You can find this kind of dependency in a broad range of mathematical algorithms. However, it is impossible to run such a loop in parallel (as written); the processor must wait for intermediate results before going on. In some cases, flow dependencies are impossible to fix: calculations are so dependent each other that we have no choice but waiting for previous instructions to complete. In different scenarios dependencies derive from the way the calculations are expressed. For instance, the above loop can be changed to reduce dependency. By replicating some of the arithmetics, we can make second and third iteration dependent on the first, but not on each other.

The number of operations has been increased – we have an extra sum – but we reduced the dependency between iterations:

Pseudocode 18 Loop example: An example pseudo code of a completely parallelizable loop unrolled

```
for i=1 to N do
    A[i] = A[i-1] + B[i]
    A[i+1] = A[i-1] + B[i+1] + B[i]
end for
```

Reducing dependency we get a slightly better performance on modern workstation, and a clear advantage on special parallel hardware.

Loop-Carried Write After Read (WAR) Dependencies This type of dependence is a whole different story than the RAW dependency. Let's loook at this code:

Pseudocode 19 Loop-Carried Write After Read (WAR) Dependencies example

```
for i=1 to N do
    A[i] = B[i] * E
    B[i] = A[i+2] * C
end for
```

In this loop, there is an anti dependency between the variable A[i] and the variable A[i+2]. We must be sure that the instruction that accesses A[i+2] reads that memory before previous instruction alters that value. Clearly, this is not a problem if the loop is executed serially, but we are looking for opportunities to overlap instructions. As we did before, it's useful to separate code and look at several iterations together.

We can directly unroll the loop and find some sort of parallelism:

Pseudocode 20 Loop-Carried Write After Read (WAR) Dependencies example unrolled

```
A[i] = B[i] * E
B[i] = A[i+2] * C
A[i+1] = B[i+1] * E
B[i+1] = A[i+3] * C
A[i+2] = B[i+2] * E -> output dependency
B[i+2] = A[i+4] * C
A[i+3] = B[i+3] * E
B[i+3] = A[i+5] * C
```

Statements 1-4 could all be executed simultaneously. Once those statements completed execution, statements 5-8 could execute in parallel. Using this approach, there are sufficient intervening statements between the dependent statements that it's possible to see some parallel performance improvements.

Loop-Carried Write After Write (WAW) Dependencies The third class of data dependencies, output dependencies, is of particular interest to users of parallel computers, particularly multiprocessors. Output dependencies involve getting the right values to the right variables when all calculations have been completed. Otherwise, an output dependency is violated. The loop below assigns new values to two elements of the vector A with each iteration:

Pseudocode 21 Loop-Carried Write After Write (WAW) Dependencies example

```
for i=1 to N do
    A[i] = C[i] * 2
    A[i+2] = D[i] + E
end for
```

As always, we won't have any problems if we execute the code sequentially. But if several iterations are performed together, and statements are reordered, then incorrect values can be assigned to the last elements of A. For example, in the naive vectorized equivalent below, A[i+2] takes the wrong value because the assignments occur out of order:

Pseudocode 22 Loop-Carried Write After Write (WAW) Dependencies example unrolled

```
A[i] = C[i] * 2
A[i+1] = C[i+1] * 2
A[i+2] = C[i+2] * 2
A[i+2] = D[i] + E <- violated WAW dependencies
A[i+3] = D[i+1] + E
A[i+4] = D[i+2] + E
```

Whether or not you have to worry about output dependencies depends on whether you are actually parallelizing the code. Your compiler will be conscious of the danger, and will be able to generate legal code – and possibly even fast code, if it's clever enough. But output dependencies occasionally become a problem for programmers.

Dependencies Within an Iteration We have looked at dependencies that cross iteration boundaries but we haven't looked at dependencies within the same iteration. Consider the following code fragment:

Pseudocode 23 Dependencies Within an Iteration

```
for i=1 to N do
    D = B[i] * 17
    A[i] = D + 14
end for
```

When we look at the loop, the variable D has a flow dependency. The second statement cannot start until the first statement has been completed. At first glance this might appear to limit parallelism significantly.

When we look closer and manually unroll several iterations of the loop, the situation gets worse:

Pseudocode 24 Dependencies Within an Iteration: unrolled

```
D = B[i] * 17
A[i] = D + 14
D = B[i+1] * 17
A[i+1] = D + 14
D = B[i+2] * 17
A[i+2] = D + 14
```

Now, the variable D has flow, output, and anti-dependencies. It looks like this loop has no hope of running in parallel. However, there is a simple solution to this problem at the cost of some extra memory space, using a technique called promoting a scalar to a vector. We define D as an array with N elements and rewrite the code as follows:

Pseudocode 25 Dependencies Within an Iteration

```
for i=1 to N do
    D[i] = B[i] * 17
    A[i] = D[i] + 14
end for
```

Now the iterations are all independent and can be run in parallel. Within each iteration, the first statement must run before the second statement.

2.1.6 Conclusion

Since the major problem in achieving better parallelization without breaking correctness is to understand the dependence relations between the statements, knowing them is a huge step forward in writing better hardware. The PM, even if only in static code, can find and represent in an exact way such dependencies.

2.2 Memory Architecture

In an usual Von Neumann architecture memory tends to be the slowest component of a computational system. Even if code can access valid memory address, practice shows that code usually demonstrated statistically access data near to previously accessed data, in a short interval of time. Based on this fact, we call temporal and spatial locality, caches have been introduced. Caches are usually much faster than RAM but have the downside of bigger silicon area, bigger power consumption and much higher cost. The idea is to put the hot data in the cache for the most part of the computation. We pay relatively high penalty if we access addresses outside the cache (called cache miss) so cores must be designed so that the working set fits in cache in order to be faster. Historically, multidimensional arrays has been layered row by row on memory where cells that differ only for the rightmost index are consecutive.

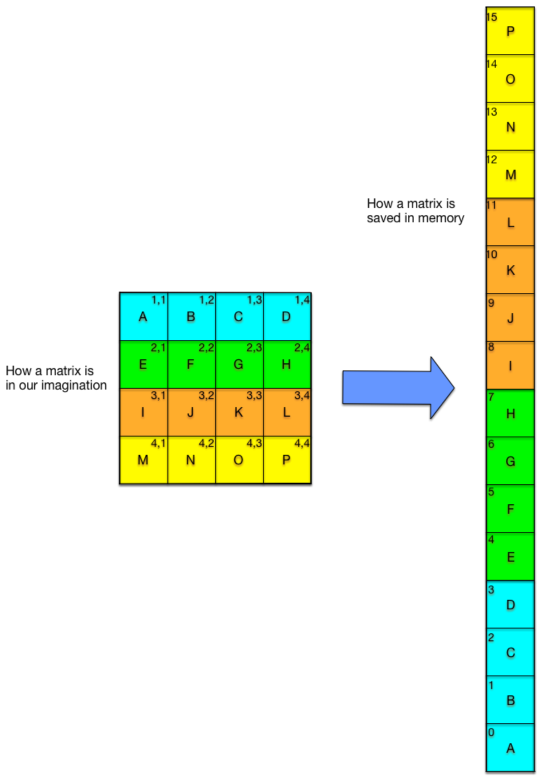


Figure 2.5:
On the left: How a matrix is in our imagination;
on the right: How a matrix is saved on memory

Hardware fetching/copy mechanism usually implements an hardware counter with starting address and number of byte to be copied. Copying chunks of data loads consecutive rows of arrays. Direct Memory Access (DMA) techniques exploits a similar concept in order to pass data between main memory and peripherals. We will pay much attention to memory accesses and reshaping memory when we later discuss how to optimize algorithms to achieve better performance, lowering communication costs. In Field Programmable Gate Array (FPGA)'s we usually have two types of memories: BRAM and RAM. BRAM resides inside the FPGA and can be configured as part of the programmable logic. BRAM usually implements a caching mechanism between functional units we will create. RAM has usually more room than BRAM, and is used to contain main data. The physical transfer between those two components happens via a synthesized DMA controller that works on the same principles I described above: base address and size. The same considerations apply both when passing data and when retrieving back from computational unit. Usually it is better to pay higher communication costs to pass down more data than needed for a single pass: doing so we allow hardware circuits to more efficiently transfer data. In fact, the higher cost paid is the setup needed for start/end the transfers. A larger transfer better amortize this cost. Similar situation happens when implementing a computational kernel on Graphic Processing Unit (GPU)s: every small stream processor has its own cache, with all the data located in the global RAM. GPU's algorithm must pay a penalty between shared memory and small caches. Additional care must be taken as data could be transferred to/from main memory accessed by the Central Processing Unit (CPU). Generally speaking, is not possible to give a general rule to modify an existing algorithm to perform better reducing memory access penalties. Even less is possible to handle every problem automatically as each problem has its own access pattern. However, if code is amenable to PA, this is less of an issue due to, for example, tiling; more on this is described in [44, 45, 46].

2.3 Related Work

As previously stated in 1.2.4, in the recent years a lot of improvements on High Level Synthesis (HLS) tools have been made in order to convert numeric or image processing algorithms written in high level languages directly into hardware implementations in order to achieve better performance and lower consumption. Thanks to High Level Synthesis (HLS), hardware synthesis requires less specific hardware knowledge in order to translate code and design hardware. HLS can dramatically speed up the development of a design: all major studies about automated synthesis show that HLS tools can speed up the generation of the hardware synthesis, too. Due to the many drawback HLS has, (such as overhead of area used and slow down in respect to a manually crafted design) studies show interests in a theoretical approach based on PA in order to achieve better synthesized circuits that HLS alone cannot generate without the knowledge and the skills of an hardware designer [47, 48, 49, 50, 51, 42, 52]. Even if PA was introduced in 60's, it is getting more and more important especially in compiler technology and in parallel computation. Using PA a compiler can achieve better data locality as well as reorganize code to split computation on more processing units [53]. On the hardware side, given the huge improvements in HLS tools, all the techniques and benefits achieved by compilers through PA can be applied on HLS as well. Current studies, such as [48, 54], exploit the PA in order to create better synthesized circuits lowering the gap between manual and automatic circuits generation.

To better explain the researches and studies that compose the state of the art, we divide the researches into different categories. The first is only theoretical and defines the state of the art in the Polyhedral Model. The latter focuses on the generation of hardware circuits or architectures.

2.3.1 Polyhedral Model

As stated in [54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67] the PM is now mature enough and has reached production compilers. In particular, in [54] authors propose a set of tools that can manipulate the polyhedral intermediate representation. Thanks to their work and the development of their tools, PM is able to deal with irregular control flow such as conditional code in loops: modern tools take in account conditional dependencies and can generate polyhedrons that correctly map and represent such constructs.

The tools proposed are:

- Chunky Loop ANalyzer (CLAN), translates all the scopes read in a file into their polyhedral intermediate representation
- Chunky ANalyzer for Dependencies in Loops (CANDL), given an intermediate representation, finds all dependencies
- Chunky Loop Alteration wizardrY (CLAY), still under development, applies all the transformations on a polyhedral intermediate representation and produces another intermediate representation
- Chunky LOOp Generator (CLOOG) [68], translates the polyhedral intermediate back into high level language

These tools have been used in a lot of other works, where the most relevant are Polly [50], Polyhedral Compiler Collection (PoCC)[69] and LooPo [70, 71, 72]. The first is a great LLVM add-on library that elaborates the LLVM-IR and applies polyhedral transformation directly on the intermediate representation before invoking the LLVM backend. The second is a collection of different compilers that can translate the input code, modify it and give back a polyhedral transformed source file.
PoCC [69] also contains other two tools for data locality improvements: LeTSeE [73] and PLuTO [53, 74]. The former performs transformation on code, achieving better data locality speeding up computation. The latter not only performs data locality transformation taking in account the L1 cache depth, but also sports

support for openMP to achieve better parallelization on multi-core architecture. LooPo is a polyhedral source to source compiler that aims in finding the best transformation for nested loop.

2.3.2 Memory Architecture

In order to maximize the reuse of data and minimize the number of BRAM usage on FPGA, a lot of work in literature has been done [75, 76, 77]. For example, in [78] they describe an algorithm to achieve better communication order in system with Sequential Communication Media (SCM) to optimize resource utilization. In [79] they focus their attention about communication cost involved in parallel computation, highlighting the importance of carefully design transfer technics and approaches on system on chip architectures. Again, in [80] they present an efficient approach for optimizing the on-chip memory allocation using loop transformations in imperfectly nested loops. Other works are described in [81, 82]. In the first one they develop an algorithm to maximize parallelism while minimize communication cost payed in loops, in the second work they analyze how important is to optimize memory usage. [83] describes how HLS can achieve 10%-30% reduction in FPGA resources and [13] is described an architecture design specifically developed for accelerate fluid registration.

A completely different optimizations where described in [77]. All of the paper briefly described before focuses on the utilization of the polyhedral model as a help to obtain better circuits. The latter, instead, focuses on HLS techniques that can optimize the synthesis with regard to current HLS tools.

The paper cited above are the most relevant ones on memory optimization, but they are not the only ones that address how optimization at memory level (or memory allocation) and memory accesses can impact on throughput.

2.3.3 Hardware Design

Before HLS came up, hardware design was very time consuming, very susceptible to the slightest error and required a lot of hardware knowledge and skill. Thanks to HLS, the designer can obtain all the generation in a far faster way. One of then main reasons to use HLS is that it can produce the right implementation for common algorithms/tasks (think about FIFO buffer for example), that is not safe to rewrite in code in Hardware Description Language (HDL). As discussed before, automatic synthesis exploits a lot of useful improvement in the development, but it pays an high cost: the used area in FPGA and the achieved performance with HLS are far worst than the manually designed counterpart. Thanks to the advancement in the PM we can manage to improve the circuits generated with HLS. The main reason behind the poor performance of automatic synthesis is that the designer must specify the various optimizations to extract the same performance of a manually designed circuits. This can be done with PM. Indeed, PM is a mathematical model, and thanks to the work described in 2.3.1, now is mature enough to be used in compilers. Thanks to this representation is possible to calculate dependencies, find bounds, reorder instructions in a completely automated way. It is even possible to infer the best optimizations an input code must have in order to be synthesized at the best with the HLS tools [84]. There are two different directions in research right now. Both of them uses the model of computation proposed earlier called PPN [85], but differ greatly in the way each research uses it. In Daedalus Framework, proposed in [86, 87], they use a tool for derivation of process network called PNGen [51]. This tool takes a source file in input and a accepts a function name as a parameter to be analized, and produces the PPN associated with all the informations about channel size between dependencies. After calculating the PPN, a tool named ESPAM [88] takes that representation and recreates the code each node has to compute. In Daedalus framework the nodes are represented as computational unit, specified in a proper file called 'platform file' that specifies the architectures used, so one or more nodes of a PPN can be mapped onto it. After the map has been completed, all the system is synthesized and streamed to FPGA using Vivado toolchain. This is a complete automated tool that starts from an high level language to hardware design without writing down a single HLS line of code. Broadly speaking, Daedalus framework maps a PPN onto a Multi Processor-System on Chip (MP-SoC). Even if it isn't directly our goal, we used PNGen to generate a PPN in order to create a network of IP Cores that communicate via FIFO buffers. We took this approach in order to explore if this solution were feasible, but we discovered it was not good for our goal. This solution was too fine-grained and consumed too much area compared to a simple synthesis using HLS directly on the "vanilla" code. We scored a huge slowdown, too. The slown-down is obviously due to the hardware we use (a Zync-7000 Zedborad) that a has fewer resources we needed in order to compare the two implementations. We also noted that the area used by our solution was too much compared to the area used by simple HLS, so we mark this solution as unfeasible.

A divergent but similar approach has been developed in [89]. In fact, their goal is to realize a MP-SoC architecture. However, how they achieve it is completely different, as one utilizes the knowledge on the code to better map the code on the processors, while the latter relies on heuristics to do so.

The other direction was pointed out in [47, 84, 48]. In their works they propose and automated framework capable of extracting the polyhedral model, restructure the code using some of the tools presented in 2.3.1 to achieve better data access and reduce area utilization. They exploit the PPN model of computation inside a single core using specific directives of HLS tools in order to generate a circuits that is intrinsically parallel. Using PM they restructure the input code so they can create a better PPN utilizing the *dataflow* directive. This directives enable the HLS synthesis to perform data flow analysis and create different circuits that can run in parallels, adding all the needed synchronization between the parallels parts created. Exploiting PM and creating a better input code is essential because it can change the way the synchronization of the tools is obtained, possibly improving performance of the generated circuits. Indeed, they achieved better performance than simple HLS, limited to a single core. What we are trying to propose is using this approach in order to speed up performance and to create a complete architecture, to be deployed even on multiple FPGA's. Using specific tools directives we can indeed generate better circuits but what about using polyhedral not only to optimize the HLS generated, but also to create the architectures capable of distributing the computation on multiple devices? Current tools do not allow to span computation on multiple cards, but PM approach is still valid to re-arrange source code. Another interesting approach about HLS synthesis is currently under development at the ECE Department of the University of Toronto, where researcher are actively developing LegUp [90]. This tool is actually a backend for Low-Level Virtual Machine (LLVM) compile infrastructure that reads LLVM-IR and generates the Register-Transfer Level (RTL) of the corresponding source code.

Apart from [90], all the work previously cited focuses on enhancing the knowledge of HLS tools, giving them specifically directives to the problem enhancing the throughput of the generated components. As described in [91, 92, 93, 94], optimization can be done at a completely different level. In fact, these papers focalize on different HLS strategies in order to synthesizes better components.

However, this optimization are on whole different level of abstraction than the work described in this thesis. This work focus on the utilization of already built HLS tools to synthesize the components, while optimize the source code used, in order to separate computation and data. No strategies on how the HLS tools performs the synthesis are made.

2.4 Tools

In this section we briefly explain the tools available in the state of the art.

Daedalus Framework

Daedalus is an open source software framework that is developed by Leiden University (UL) and University of Amsterdam (UvA). Daedalus provides a single environment to create a "system-level architectural exploration, high-level synthesis, programming and prototyping of multimedia MP-SoC architectures". Daedalus aims at the creation of a system-level design going from a sequential application to a working MP-SoC prototype in FPGA technology. Deadalus could offer great potentials for quickly experimenting with different MP-SoC architectures and exploring design options during the early stages of design.

PLuTO

PLUTO is an automatic parallelization tool based on the polyhedral model. The polyhedral model for compiler optimization provides an abstraction to perform high-level transformations such as loop-nest optimization and parallelization on affine loop nests. Pluto transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient tiling and fusion, but not limited to those.

LetSeE

LeTSeE is a platform dedicated to computing and exploring the legal affine scheduling space of a statically controlled program. It has been built up as a library, offering services such as:

- a tunable algorithm for legal transformation space construction,
- various heuristics to traverse legal spaces,
- many auxiliary functions (graph manipulation, transformation generation, etc.)

Chunky Loop ANalyzer

Chunky Loop ANalyzer (CLAN) is a tool to extract the polyhedral representation from the Static Control Parts (SCoP) of high level programs (written in C, C++, C# or Java). It is an in-development tool, but at this state is capable of extracting almost all polyhedral representation of scientific programs. This tool is based on the LLVM compiler infrastructure in order to validate the syntax of the input code. CLAN analysis is derived from Clang [95] libraries.

Chunky Loop Alteration wizardrY

Chunky Loop Alteration wizardrY (CLAY) is a tool to apply high-level loop transformation scripts to Static Control Parts (SCoP). It accepts all major loop transformations (fusion, fission, skewing, interchange, tiling, unrolling etc.) as well as data transformations. It is able to check for the legality of the transformation script as well as generating the code that implement this transformation script.

Chunky ANalyzer for Dependencies in Loops

Chunky ANalyzer for Dependencies in Loops is a tool for data dependence analysis of SCoP. This tool take in input the polyhedral representation of a SCoP and output a new polyhedral representation that includes all the dependencies domain between statement.

Chunky LOOp Generator

Chunky LOOp Generator (CLOOG) is a code generator for scanning Z-polyhedra: it finds the code or pseudo-code where each integral point of one or more parameterized polyhedron or parameterized polyhedra union is reached. In order to pass the PM between tools Chunky Loop ANalyzer, Chunky Loop Alteration wizardrY, Chunky ANalyzer for Dependencies in Loops and Chunky LOOp Generator (CLOOG), it uses the OpenSCOP specification [96] data format.

Vivado Design Suite

Vivado is a design suite developed by Xilinx aimed to speed up the generation of hardware design of FPGA's. It is composed of different tools:

- Vivado HLS: it reads C/C++ source files and generates the corresponding RTL
- Vivado: it generates the bitstream reading tcl files, or composing manually the design
- Software Development Kit: it allows to issue commands to the hardware subsystem in cards, such as processors, DMA controllers, network cards and synthesized components

LegUp

LegUp is an open source high-level synthesis tool being developed at the University of Toronto. The LegUp framework allows researchers to improve C to Verilog synthesis without building an infrastructure from scratch.

2.5 PA and HLS Limitations

Even if the current state-of-the-art is very promising we must point out the following limitations:

- HLS requires deep knowledge of optimization directives and mutual relationship, for example *unroll directive* in combination with *pipeline directive* can lead to create a circuits that goes beyond physical resources of an FPGA. We cannot automate this behavior in current HLS tools.
- PM can process and transform only affine code: even promising approaches such as "weakly dynamics" [97] extend the domain of the PM to data values, they cannot manage a whole class of algorithms based on recursion, pointer arithmetic, aliasing, generic lists and so on.
- All current toolchains focus on single FPGA design. Most of them consider only optimization on a single circuits. No one considers take in account architectures based on multiple cards, mostly due to synchronization and data exchange issues.
- HLS tools consider only a single card at time and produce bitstream ready to be deployed only on that one.
- Current toolchain cannot predict the amount of area the algorithm will occupy without relying on HLS tools. This try-and-error approach is not viable.

3

Problem Statement

In the previous Chapters 1 and 2 I presented the State-of-art and listed all the major limitations of the approaches and tools currently employed. In this Chapter I formally express the problem statement of this thesis.

3.1 The Problem

A lot of research focused on the automatic design of hardware components by means of High Level Synthesis (HLS) [98, 99, 100, 80]. Sometimes it is possible to find works aiming at the parallelization of specific scientific algorithms. Too often, the generation and adaptation of these algorithms require a lot of time and a deep knowledge of the computation to generate an efficient hardware version on an FPGA. As the computational horsepower required by the scientific community rise year after year, we need to find a way to cope with demand with more mature tools and power efficient solutions.

3.2 Problem Approach

Typical scientific algorithms are written (or can be or can be easily adapted) in a mix of static, affine and pure code; for this, we can explicitly use the Polyhedral Model (PM) to model, transform, and parallelize these types of workloads. As stated in [101], the polyhedral optimization framework has been demonstrated as a powerful alternative to abstract-syntax-tree based loop transformations. As such, the code can be better manipulated and enhanced for performance using Polyhedral Analysis (PA). This can definitely improve current HLS tools. In fact, a limit of actual HLS tools is the inability to infer the best directives to use in order to generate the best circuits possible.

However, the optimization on the components generated by the HLS tools is only a minor part of the process. Since these kind of algorithms are heavily data-parallel, we cannot utilize only one custom component to speed up the computation. Instead, in order to enhance the throughput, we need to rely on multiple hardware cores, each of them computing on different sets of data. The same vision must be kept when single card solution is no more enough to satisfy problems that go beyond resources (in terms of amount of data and/or computational load): in such case the approach must be the same but scaling and adapting to a new configuration implementing multiple cards, and taking into account all synchronizations, data sharing/exchanging and time constraints.

3.3 My contribution

The challenge addressed in this essay is *finding a methodology that can divide computation between different isolated sub-kernels to obtain the parallelization of scientific workloads*.

In literature we can read a lot of approaches that use PM to model to achieve advanced parallelization features and performance, often using e HLS tools: the contribution can be considered innovative as the goal is to divide computation on multiple sub-kernels in a way that is independent from the hardware, and reserving the choice of specific cards / hardware features at a later stage, trying to keep it as independent as possible from specific features, allowing to deploy the kernels on range of hardware solutions as wide as possible.

As the target of this work is to modify the source code at a higher level, the process must generate code that depends on hardware resource availability only in the later stages of my toolchain.

Only HLS tools can set constraints about the amount of HW resources: the toolchain must be notified about these constraints and eventually can re-configure on more discrete cards, or signaling that the physical implementation cannot satisfy the algorithm. The manual configuration/intervention of the user must be kept as low as possible, delegating all the decisions about transformation and synthesis to the toolchain. One of the key aspect of this work, is the possibility to split computation on multiple devices. *Current HLS tools cannot do this without explicit System Level Design (SLD) input from users*. This is not a simple matter of physical resources/layout settings: this estimate involves theoretical considerations about models of computation which is out of the scope of HLS. But thanks to the PM this heavy work can be done automatically.

One of the peculiarities of PM is the ability to know every dependence from the source code. Even if the price to pay is working on relevant but relatively limited sets of algorithms, this feature is too good to overlook. In fact, if we have the power to automatically extract the dependence knowledge from the code, we are also capable to infer the best parallelization cut to apply on the code to create most independent sub-kernels.

Even if with that knowledge on some algorithms is not possible to choose a cut without side effect, since we have the complete understanding of the flow dependencies, we can also manage and handle them within parallels sub-kernels.

3.4 Delimiting the Perimeter of Interest

This work is the first step to validate the convenience of the automated approach presented so far. Since the beginning we chose to focus on the automatic generation of an architecture starting from a scientific algorithm, in particular we focused on "stencil code" that typically manages arrays using invariant small computational kernel. This kind of algorithm can be expressed very well with static code as data can be accesses using only indexes on a regular pattern. So PM can achieve great results as mutual dependencies on data are typically low and access can be precomputed at compile time by automatic tools. Thus we can extrapolate huge amount of parallelism via polyhedral transformation of the code. Obviously, the PM was built with software in mind (i.e. shared memory, multi-thread/multi-process), so we need to adapt the PM tools taking in account physical characteristics of hardware circuits. Consider, for example, how physical circuits access matrices kept in RAM: PM tools generally do not consider row vs. column access. We can reduce significantly access/exchange times leveraging on typical row by row access. Note these tools such as PLuTO (see 2.3.1) can perform optimization considering hardware details (L1 cache) improving data locality, but don't take in account physical layout of data. Generally speaking, an algorithm written keeping in mind hardware data layout can achieve far better performance. Keeping in mind previous consideration, the perimeter has been delimited as follows:

- Computation must be split using PM
- No need to rewrite PM tools from scratch
- Results from PM processing must be ready to be synthesized independently on discrete boards using HLS
- Some algorithms need to be rewritten to extrapolate better parallelism

To summarize I will focus on checking the structure of algorithm, deriving functional dependencies using PM tool (Chunky Loop ANalyzer (CLAN) and CANDL), divide tool output in discrete files leveraging on automatic splitting yet performed (CLAY) and generate Tcl files needed to HLS tools.

While each step in this work is done **manually**, it is important to note that they can be easily **automated**. The creation of the PM, the extraction of the dependencies and the regeneration of the transformed code can be done **automatically** since they are techniques explained in state-of-art polyhedral compilers. The most difficult part to automate is the identification of the best splitting cuts as it requires design space exploration techniques; while this is the most relevant manual step, it could be **automated**, too.

As we will see in next chapter, I had deliberately reduced the perimeter to a substantially manual approach as results derived from a fully automatic toolchain we experimented, brings to inefficient solution in terms of too many conditional statements (even if needed) and thus to huge area usage and slow down, even in software simulations.

4

Proposed Methods

In this chapter I will explain the major hardware constraints of the problem that needs to be taken into account in order to exploit better performance from circuits created with High Level Synthesis (HLS). Also, I will explain the main ideas and procedures that brought me to the creation of the first toolchain and the final methodology, and the way, in which quasi linear speed up is achieved, along with separation of data and computation.

4.1 Hardware Constraints

Besides algorithmical and logical considerations, the real constraint that affects problem domain is due to tight hardware limitations. Any solution the t:he problem must respect resource constraint. The two most constrained resources in data parallel, compute intensive application are:

- DSP slices
- BRAM blocks

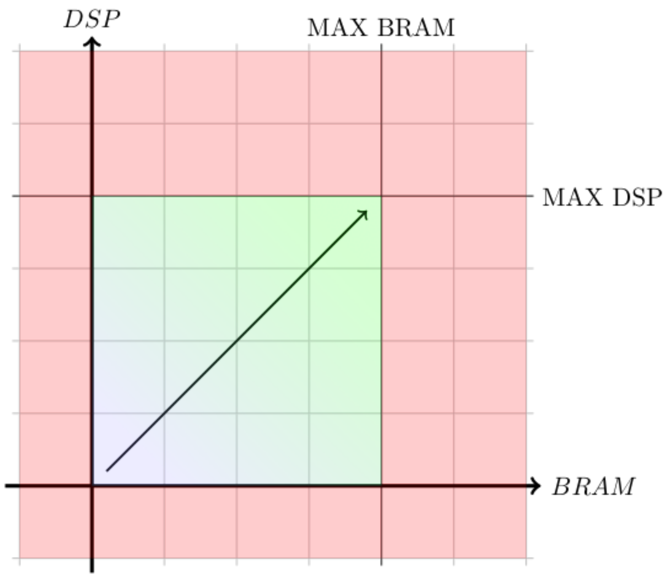


Figure 4.1: Direction of BRAM and DSP usage.

4.1.1 DSP Slices

DSP functional blocks usually implement mathematical computation. There is a strong dependence between elementary arithmetic operations and number of DSPs that must be synthesized on Field Programmable Gate Array (FPGA). The percentage of DSPs usually has positive effects on computational speed without sacrificing other types of functional blocks (i.e. BRAM, LUT). Referring to figure 4.1, it is better if we can move the design in the upper part of the diagram. Given a workload, maximize DSPs usage is beneficial and, power efficiently, as the resulting equivalent functionalities in LUTs slices increases latency, critical path and energy required to move data (i.e signals) between less specialized components.

4.1.2 BRAM Blocks

BRAM blocks are dedicated to store data values. This is the premium resources on FPGA. BRAM usage grows with the size of locality stored data. Even minimal savings in BRAM usage allow us to:

- Implement advanced directives (for example data flow)
- Redistribute area to single components, allowing them to process more data per single transfer from-to cores

4.1.3 How Advanced Directives Shape Design Space

Any approach about parallelism and hardware synthesis cannot leave aside three important technics:

- Loop Unrolling
- Pipelining
- Dataflow Pipelining

Xilinx Vivado HLS [102, 103, 104, 105, 106] allows to specify three special directives in order to enhance throughput. These directives resemble the above technical aspects:

- set_directive_unroll (Loop Unrolling)
- set_directive_pipeline (Pipelining)
- set_directive_dataflow (Dataflow Pipelining)

Every single directive can greatly modify resulting design by varying area occupation, number of DSP's and BRAM's involved and so on. Let's consider in detail all the above directives:

set_directive_unroll (Loop Unrolling)

Unroll can be implemented into multiple stages of the process: in Polyhedral Model (PM) or in HLS tools directives. The final effects are essentially the same:

- Huge speed up in throughput
- DSP's slices are exhausted very quickly
- In some cases frees BRAM blocks (lesser need of intermediate buffers)

Notably the utilization of DSPs really depends on the HLS strategy adopted by the tool. All of the above statements are made using a strategy that tends to use DSP slices for calculation rather than LUT's. In the following example is shown the unroll directive functionality:

Pseudocode 26 Unroll: An example pseudo code of a normal code

```
for i = 0; i < N; i++ do
    A[i] = B[i] + C[i]
end for
```

If we apply unroll the directives, said with a factor 4, the resultant code is:

Pseudocode 27 Unroll: An example of an unrolled code

```
for i = 0; i < N; i+=4 do
    A[i] = B[i] + C[i]
    A[i+1] = B[i+1] + C[i+1]
    A[i+2] = B[i+2] + C[i+2]
    A[i+3] = B[i+3] + C[i+3]
end for
```

HLS tools generates a component capable to run all four statements in parallels, hence enhancing the throughput, but at the cost of a 4x area used.

set_directive_pipeline (Pipelining)

Enabling pipelining can enhance performance as removes serialization schedule that HLS will produce by default, showed in figure 4.2, and produces a more parallel schema as showed in 4.3. This directive increases the use of DSPs so the design schedule becomes that of in 4.1.1. Impact on BRAMs usage can vary, but in general downs't free them.

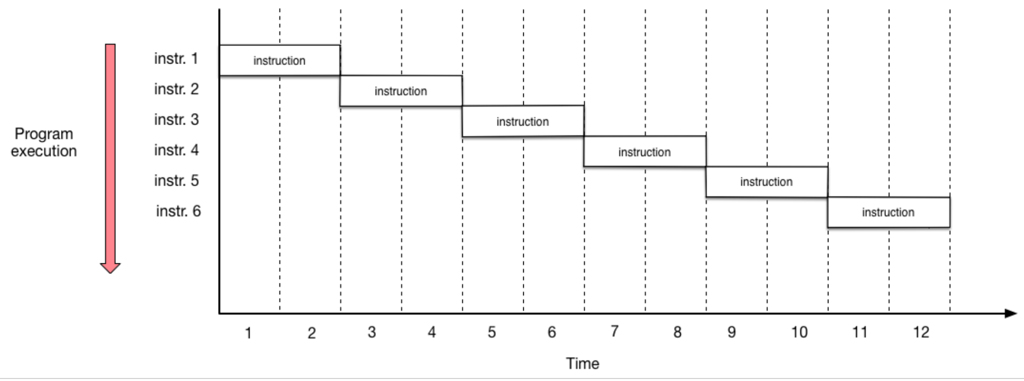


Figure 4.2: fig:Execution flow without pipelining

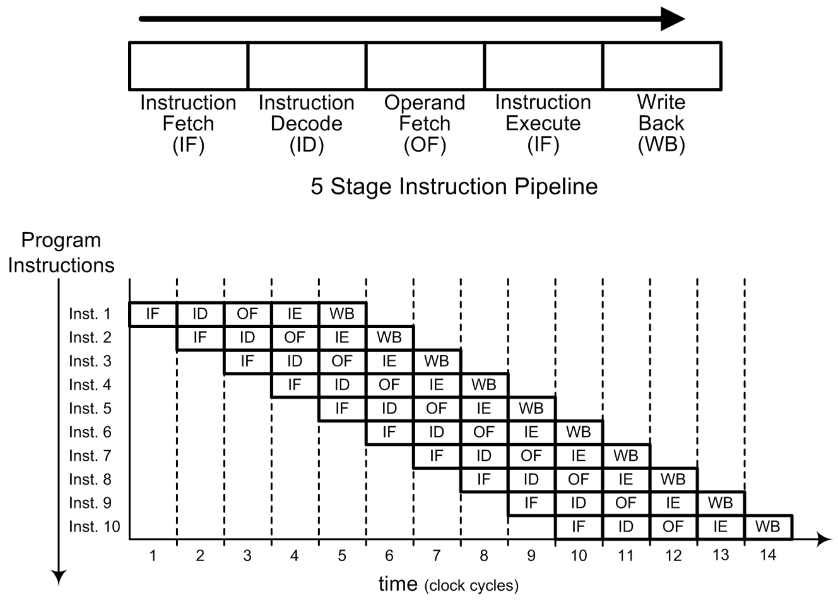
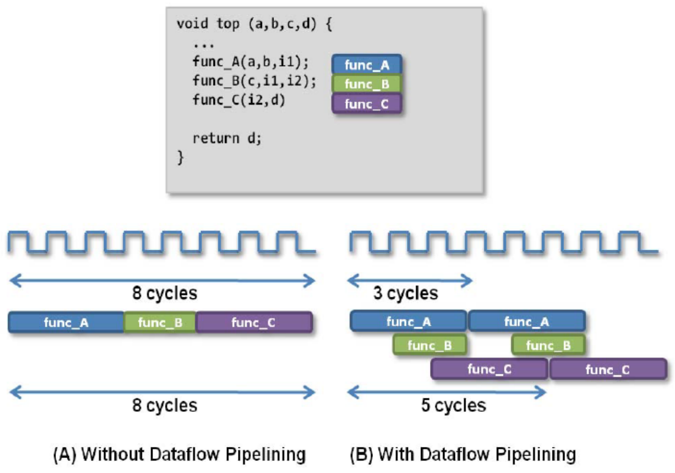


Figure 4.3: fig:Execution flow with pipelining

set_directive_dataflow (Dataflow Pipelining)

Dataflow directive enhances throughput by triggering the creation of parallel blocks of functional units inside the same core. It can increase performance at the cost of larger BRAMs usage. In fact, this directive inserts memory buffers between functional blocks in order to preserve the correctness of the original computation.



```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

Figure 4.4: Dataflow Directive Behavior

Summary

To summarize, all these directives are fundamental part of work setting as they can hugely improve the speed up and the resource utilization towards optimal throughput/consumption ratio.

However, no optimization at the HLS strategies level will be made to further improve the capabilities of this directives.

In the next sections I will explain what I have done towards the goal of the thesis.

4.2 First Approach

The first project I looked at is the Daedalus framework [86]. It proposes an automatic toolchain capable of creating an architecture starting from a high level language, exploiting both PM and reconfigurable hardware. This toolchain aims at the creation of an Multi Processor-System on Chip (MP-SoC) based architecture. I relied on this method to represent the flow of the computation, extending

the toolchain with custom cores.
After analyzing and testing Daedalus, I decided it was worth exploring a similar direction using the PNGen tool [51], contained in the Daedalus toolchain. PNGen is capable of generating a Polyhedral Process Network (PPN) of a specific function (i.e. C/C++ functions) from an input file written in high level language. Given the model of computation created by PNGen, I focused on using this polyhedral representation to meet my goal. I developed a tool capable of taking as input the PPN generated by PNGen and producing the corresponding files needed by HLS tools. As I started from the same idea of the Daedalus framework, each node in PPN becomes an individual custom core, and every communication between nodes are translated to FIFO buffers. The final product is an automatic toolchain that generates the hardware architecture files just ready to be synthesized by the Vivado toolchain, and is briefly described in figure 4.5.

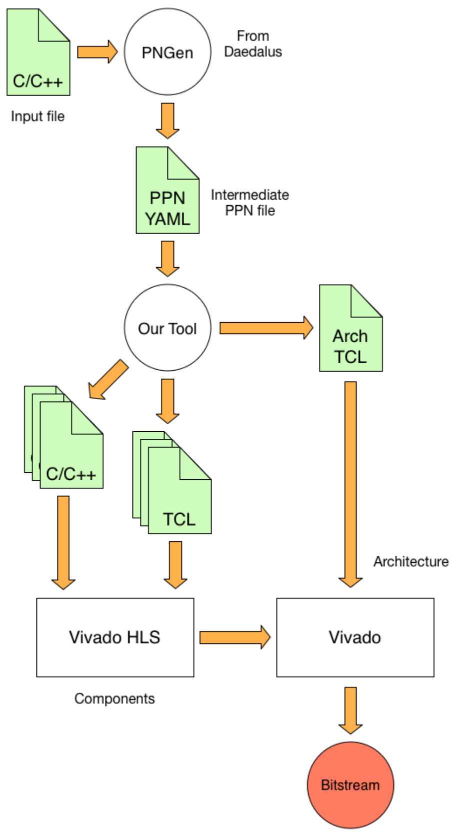


Figure 4.5: First Toolchain

4.2.1 Components

The C/C++ file must be written with the limitations imposed by PM enunciated in the previous chapters. This tool outputs a YAML file, containing the topology of the corresponding PPN of the source file. I replaced the Daedalus framework tool ESPAM with a custom tool that builds a different architecture files. In fact, what I wanted wasn't creating an MP-SoC based design, but generating a dedicated hardware modeled on the PPN previously computed by PNGen. So my tool takes as input the topology YAML file generating the corresponding C++ source and the Tcl script files ready to be synthesized. The other tools are Vivado HLS and Vivado, which are necessary to perform the generation of the RTL and bitstream automatically, since we target Xilinx FPGA's.

4.2.2 Flow

Toolchain flow is pretty linear. As described in figure 4.5 and in subsection 4.2.1, the flow starts with a C/C++ file, containing only affine code, as input to the PNGen tool. The tool I developed takes the output of PNGen, a YAML Ain't Markup Language (YAML) file describing the topology of the PPN and outputs all the C++ and tcl files needed to generate the architecture. Using Vivado HLS than I synthesized the accelerated cores, and with Vivado I connected them and finally generate the bitstream, automatically. The software part, developed with Xilinx SDK, related to the initialization of the design and the exchange of data between main memory and the FPGA, is done manually. As previously stated, my tool is developed to replace the ESPAM tool, part of the Daedalus toolchain. This is exactly the point of divergence of my toolchain from Daedalus. I do not want to develop an MP-SoC architecture: what I want is to generate directly the hardware kernels needed for the computation.

4.2.3 Limitations

The toolchain is completely valid, functional and mostly automatic. However, the first toolchain suffers from two huge problems:

- Huge area used
- Huge slow downs

These two problems derive from the construction of PPN performed by PN-Gen. Since this PPN is composed of nodes, each containing only one statement, HLS tools are unable to optimize area and resources. Another reason of the bad area utilization is the huge numbers of dependencies. Dependencies are translated into FIFO buffers, that must be enabled only on particular conditions, so we need *multiplexers* to implement the conditional logic. The usage of FIFO buffers and multiplexers at once results in great amount of resources spent. Another side effect of buffers and multiplexers is the creation of multiple *critical paths* even at low frequency. However, the main reason of these drawbacks is the too fine-grained logic network produced by PNGen.

As my experimental results demonstrated bad overall performance, whether considering area or throughput, I revised the whole work in the light of what has emerged during the tests. This considerations had an important role in the directions to take in further developments, but I will no more consider this toolchain in the following of this work.

This has led to the final toolchain I am about to describe.

4.3 Final Approach

Taking in account all the limitations and issues arisen from the experiments explained above, I started from a simple consideration, supported by experimental results: algorithms directly synthesized, without relying on the first toolchain performs much better than the polyhedral counterpart. This is due to an incorrect usage of PM: the too fine-grained approach of PNGen creates too many channels between hardware implementation of software statements, separate dependencies, adds to many controls. On the other hand, the directly synthesized components don't need so many channels, as all the communications are directly implemented by generated HLS.

Since the goal is to parallelize computation, the main idea is to create as many components as FPGA bears, each of them computing the same operations. However, those components will not use all the data, but only on a restricted subset to enhance throughput.

As a result, I came up with the following methodology.

4.3.1 Components

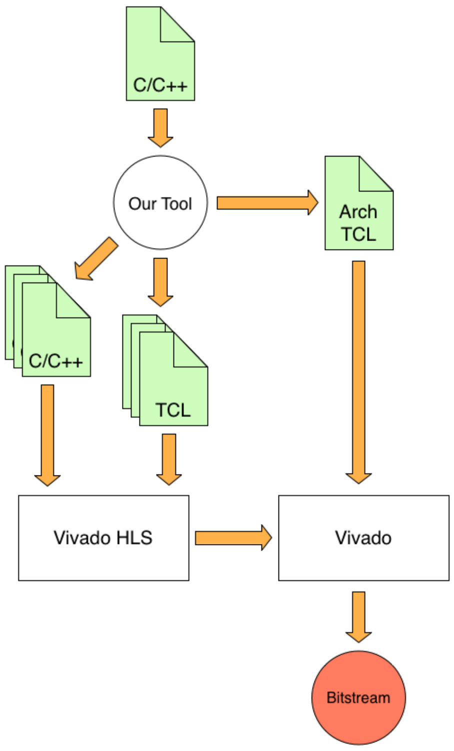


Figure 4.6: Second Toolchain

In Figure 4.7 is described the toolchain scheme. It is very similar to the one proposed in the first toolchain but has one less stage since we drop the usage of PNGen to extract the PM. Since I stopped relying on PNGen, I needed to take one step back in order to extrapolate the PM from the source file. What I have done is to directly employed the tools used so far, and look at how they generate and manipulate the PM. This led me to the discovering of four major tools, some of which were used inside PNGen, too, and other PM tools described before. As a reminder, I will list them here:

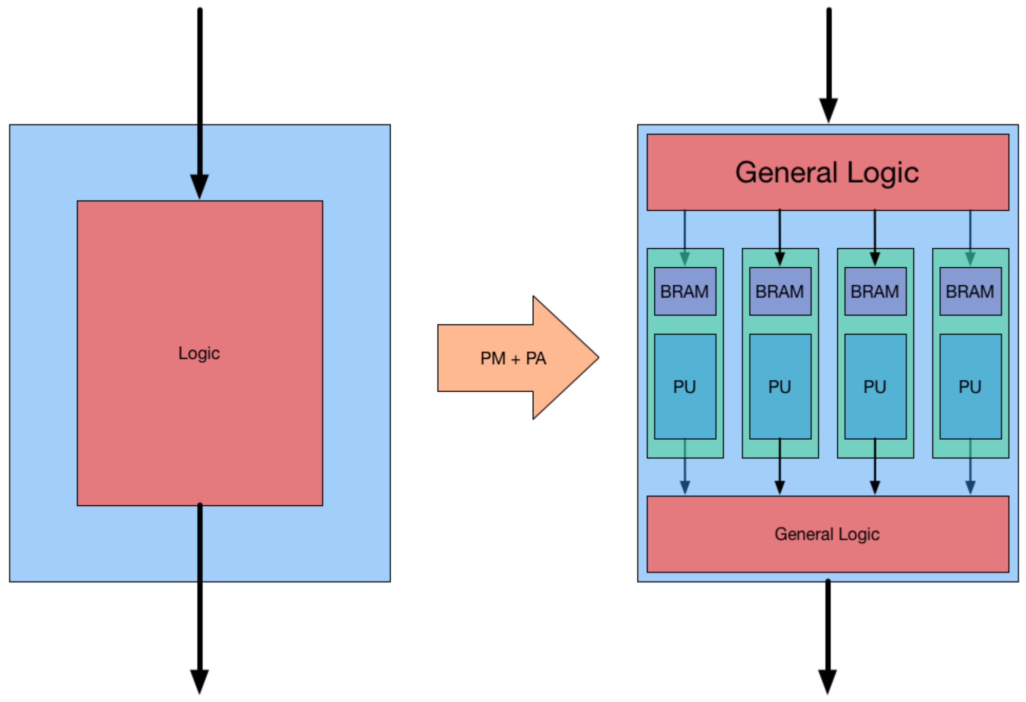


Figure 4.7: Left: Components synthesis without PM optimization; Right: Components synthesis with PM optimization

- Chunky Loop ANalyzer (CLAN) (2.4)
- Chunky Loop Alteration wizardrY (CLAY) (2.4)
- Chunky ANalyzer for Dependencies in Loops (CANDL) (2.4)
- Chunky LOOp Generator (CLOOG) (2.4)

4.3.2 Methodology

The current developed methodology is fully working, although, as of now, all the intermediate steps are done manually. However, all the passages can be made automatic.

In fact, once the source code is transformed in the polyhedral representation, everything is known about that code. From a theoretical point of view my code transformations does not differ from transformation a compiler performs to optimize the code. I will start using the statement representation present in polyhedral model generated by CLAN. Note that a similar approach is used in CLOOG.

An important step outside of the methodology is to rethink the original algorithm to implement parallelism. This is not mandatory as our toolchain can process unmodified code, but polyhedral tools I use will not be able to extract the disjoined domains I need to produce efficient parallelized code. In fact, if you process an algorithm written without having parallelization in mind, the polyhedral tools will produce worse code due to the huge amount of flow dependence (see chapter 2.1.5). This initial set up cannot be automated as we explained in theoretical considerations expressed in Chapter 2. The core idea is to avoid separating each statement in single computational unit as I did writing the first toolchain, since this approach introduces a great overhead and slowdowns preventing synthesis tools to exploit advanced dependence analysis optimizations. Also, this approach aims at the creation of a parallelized version of scientific algorithms. So, I want to generate multiple sub-kernels that are capable to compute on less data, favoring parallelism. Obviously, this division in multiple pieces will introduce some communication costs, but as I will show in the next chapter, these costs are overcome from the huge throughput gained. The important aspect is that all of the optimizations that can be introduced with the help of Polyhedral Analysis (PA), are completely orthogonal from HLS directives.

Pseudocode 28 Pseudo code of Second Methodology tool

```
$pmRepr \leftarrow callClan(InputFile)$
**if** pmRepr != regular **then**
  $exit(-1)$
**end if**
$deps \leftarrow callCandl(pmRepr)$
$deep \leftarrow findParallelDeep(pmRepr, deps)$
$inputTransf \leftarrow$ compute the best split
$transformedPM \leftarrow callClay(pmRepr, deps, inputTransf)$
**if** transformedPM is not valid **then**
  $exit(-1)$
**end if**
$cFileNames = []$
$tclFileNames = []$
$tclArchitectureFileName$
**for all** Scop **in** transformedPM **do**
  $listBlocks \leftarrow getBlocksAtDeep(pmRepr, deep)$
  **for all** block **in** listBlocks **do**
    $cFileNames.append(writeCFile(block))$
    $tclFileNames.append(writeTclFile(block))$
  **end for**
  $tclArchitectureFileName = writeArchitectureTclFile(listBlocks)$
**end for**
```

Note: usually there is only one Static Control Parts (SCoP) so the outer Scop iterator can be removed.

Before starting to analyze the pseudo code 28 and all the transformations involved, we need to understand what *split iteration domain* means.

Split Iteration Domain If, for example, we want to split the code shown below, we need to separate the whole domain of the iteration in sub-parts.

Pseudocode 29 Example of nested loops

```
1: for t=0 to TSTEPS do
2:     for i=1 to NI-1 do
3:         for j=1 to NJ-1 do
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:         end for
6:     end for
7:     for i=1 to NI-1 do
8:         for j=1 to NJ-1 do
9:             A[i][j] = B[i][j]
10:        end for
11:    end for
12: end for
```

If we look at the dependencies, it is evident that we have a flow dependency between the statement on line 4 and the one on line 9. Additionally, there is a loop-carried flow dependency between statement on line 9 and statement on line 4. So we cannot make any split on the domain of the first loop (loop with index t). So, if we want to parallelize this code, we need to look deeper in the nested loops. If we eliminate the outermost loop, the resultant code will be more easy to parallelize as the loop carried dependency has been removed, so we can conclude that the inner block of instructions can be parallelized. It will appear clear after looking at the polyhedral representation of the domain iteration.

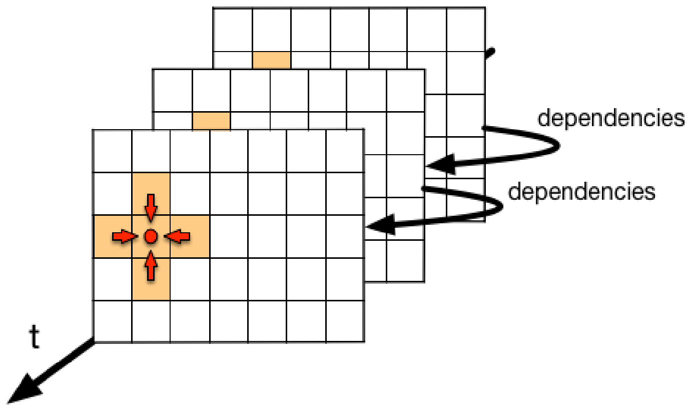


Figure 4.8:
Representation of the dependencies
between different iterations
of the outermost loop

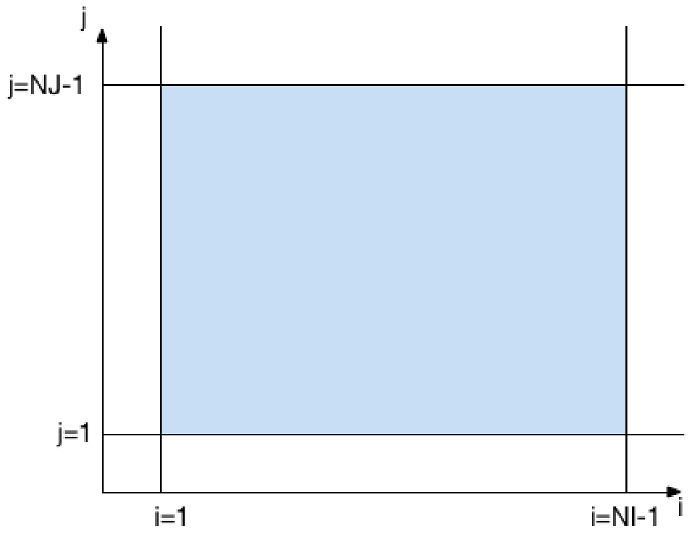


Figure 4.9:
Iteration domain of the block
inside the outermost loop

Between one iteration and the others there is no dependency so we could, for example, divide the iteration domain and separate the computation in two sub, independent parts, as in the following figure.

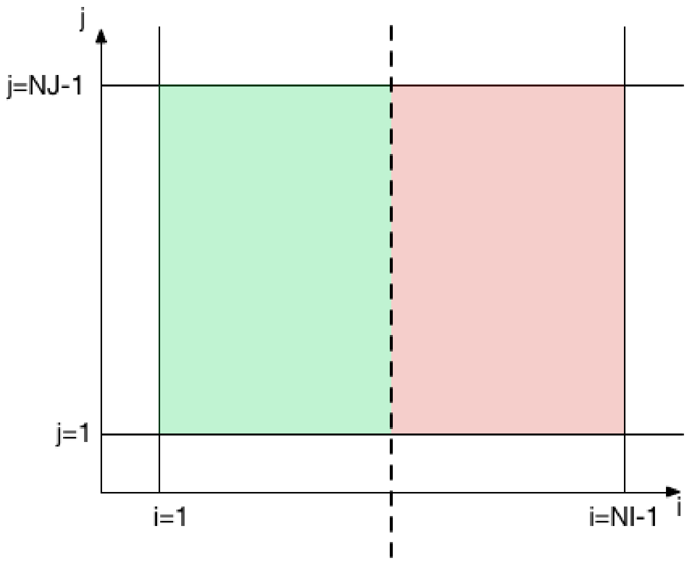


Figure 4.10:
Split of the iteration domain
of the block inside the outermost loop

We can create two computational kernels, the first that computes the green part, and the second that computes the red one.

Pseudo code 28 above describes the essential pass designers should follow in order to create the architecture. After selecting the code to be analyzed, CLAN must be invoked in order to get the PM of the source code marked by pragmas. Once the polyhedral representation has been generated for that SCoP, the next step is to extract knowledge about the dependencies which is performed by CANDL. Since we have the knowledge of both dependencies and polyhedral model, we can iteratively find the nesting deep in which there are no flow dependencies(Read After Write (RAW)) between that loop and the outer ones and split it with CLAY. Even if PA is able to perform various transformations on the code, for my purpose I choose to rely only on the split domain transformation as the main goal is to create an highly parallel architecture. After the new polyhedral model representation is created by CLAY, we can pass it to CLOOG and generate back a polyhedral transformed C code. The split code is separated in different files as I need to create different IP cores from different sub-kernels. Using CLOOG I automatically generate the source and tcl files needed for HLS tools, creating the tcl file needed for the architecture generation, too. After all these steps have been done, the global toolchain will call Vivado HLS for every pair of (cFileNames[i], tclFileNames[i]) and in the last pass the toolchain will run Vivado passing the architecture tcl. In the end, the resulting architecture will be composed by a processor and an AXI DMA[1] controller for every accelerator. All the communication are done utilizing the AXI4Stream[2] interface protocol.

Due to the creation of independent components, the generated architecture can be easily ported on a multi-FPGA environment with relatively little effort: we can map kernels with no restriction on a specific FPGA using a similar approach to the aforementioned described. A very different issue is not on the theoretical side but will be the real hardware implementation in design tools were we need to add all the necessary hardware glue between FPGAs.

[1]AXI DMA: An advanced DMA transfer technique available on FPGA that adheres to AXI protocols.

[2]AXI4Stream: An advanced transfer protocol used to implement streaming functionality between hardware components.

In this methodology no memory optimization has been done utilizing PM. As I already mentioned, this optimization can extend to my methodology and can be added later without compromising the validity of the work.

5

Experimental Results

In this Chapter I will present the results obtained using the final methodology described in the previous Chapter.

5.1 Practical Examples

The kernels we tested came from the Polybench suite [107]. Some of the kernels had to be slightly revised in a more FPGA-friendly way (i.e. adopting smaller arrays, adding channels and so on). These tests are made with the aim to validate the methodology in order to explore further optimizations in the future.

It's not a secret that the Zedboard isn't so spacious. Let's admit that using a ZedBoard is substantially different than using a real Field Programmable Gate Array (FPGA), given the low frequency achievable on very complex design, but once we get good results on it, we are assured we can move to real FPGA's with the same logic approach and scalability, getting far better results. Let's consider, for example, a Virtex-7: it can bring a lot more resources, a more advanced transistor generation and a bit higher clock frequency. Hence, apart from some differences in efficient energy usage, they are not so different. In figure 5.15 we show how we expect the result on a Virtex-7 should be. The linear behavior shown on the Zedboard is preserved as the architectures will be the same, but since the Virtex-7 has more resources available is possible to generate more parallel cores and so speed up further the computation. Also, since the Virtex-7 is able to create a circuits with higher frequency than a ZedBoard we should see an even higher throughput. Now, that we can assume that a prototype on a smaller FPGA is valid, we can come back to the parallel implementation of the kernels.

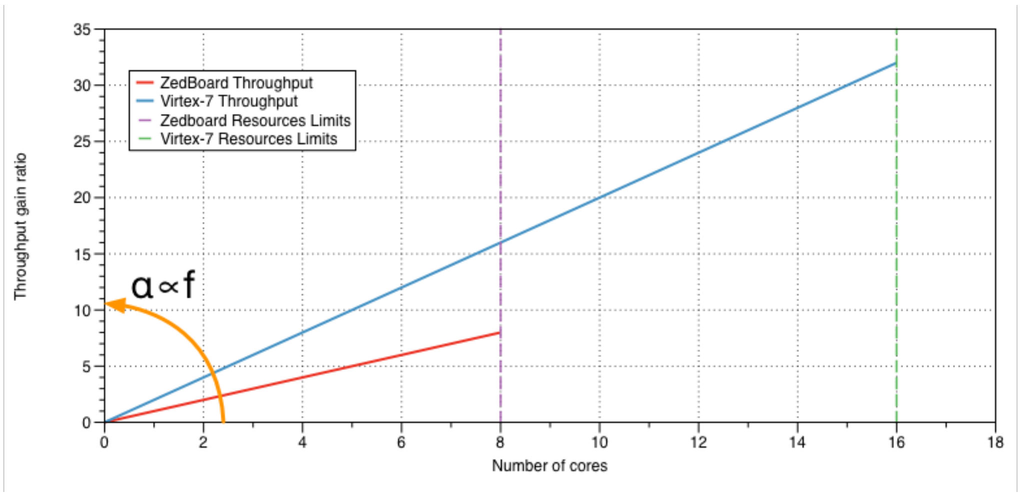


Figure 5.1: Gain Ratio Virtex-7 vs ZedBoard

We tested the following kernels:

- Jacobi 2D stencil computation kernel
- 2mm, two matrix multiplication kernel
- 3mm, three matrix multiplication kernel
- 2-D convolution kernel
- BiCG Sub-kernel

5.1.1 Jacobi 2D stencil computation

Th Jacobi 2-D stencil is an iterative computational kernel that, taken in input a matrix, computes the mean value on five points accessed in a cross shape pattern for every elements as described in figures 5.2. It is the kernel for many linear algebra and image analysis algorithms.

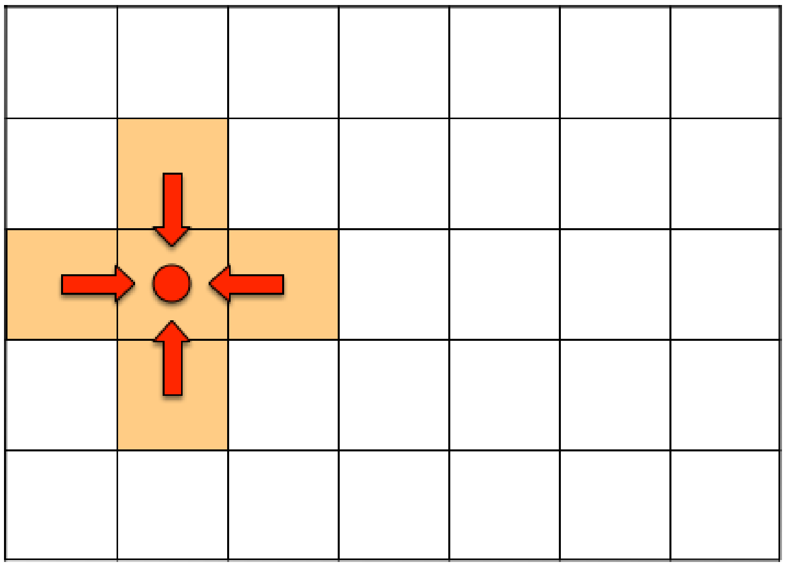


Figure 5.2: Jacobi 2-D Computation

Let's just briefly analyze the code below. The code is composed of an outer loop, that we call iteration loop, and of two blocks each containing two nested loops. The first internal block computes the mean value and the second updates the matrix. The outer loop works on multiple passes getting the convergence after some iterations. Obviously, since the computation of the mean value and the matrix update are done in a sequential way, these two blocks cannot be run in parallel. In fact, as it is clear from the following code, statement on line 7 and on line 12 are dependent: specifically, there is a Read After Write (RAW) dependence between the two. The other dependence in the code is a RAW loop carried dependence (see 2.1.5) between the same two statements, but now the directions are reversed. This dependence limits the parallelization that can be made on the code, but also help us in building up the parallel kernels. Indeed, if we look at the computation statement (line 7) it is clear that each iteration doesn't depend on the others. So the two operations in both blocks can be separated in order to enhance the throughput.

Pseudocode 30 Jacobi 2D stencil sequential code on 300x300 matrix

```
 1: #define NI 300
 2: #define NJ 300
 3:
 4: for t=0 to TSTEPS do
 5:    for i=1 to NI-1 do
 6:       for j=1 to NJ-1 do
 7:          B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][1+j] + A[1+i][j] + A[i-1][j])
 8:       end for
 9:    end for
10:    for i=1 to NI-1 do
11:       for j=1 to NJ-1 do
12:          A[i][j] = B[i][j]
13:       end for
14:    end for
15: end for
```

So, if, for example, we change the first parameter NI from 300 to 75, (exactly a quarter of the original computation) it is possible to run four different kernels to compute the mean value and update the matrix. At this point one can argue that the loop carried dependence prevent us from doing so. This is correct, but to an extent: in fact, the correct number to split the computation is not exactly 75, due to the border effects this type of computations produce. The correct numbers in this case is 76 for the first and the last block and 77 for the two middle blocks. This will be true if we split the computation on the matrix only on the row dimension. This approach can be generalized for every number of splits. Obviously, this is not the only cut we can produce: it is possible to split the computation by columns, or create squares. It is even possible to cut with oblique lines. When we come to implementation, unfortunately, the data access pattern and transfers between hardware units becomes expensive. So, the best performances are achieved cutting down the row dimension, as doing so we pay a lower communication cost. Due to the loop carried dependence coming from to the iteration loop, it is not possible to re-organize dependencies further. In this case, once every computation inside an iteration is completed, before starting the next iteration we must update the whole matrix or create dedicated channels between kernels in order to update the data. We chose to opt for the straight solution, where we update at each iteration the whole matrix. Note that this approach, limits neither its validity, nor the scalability: our current and most important goal is to demonstrate that Polyhedral Model (PM) can be used to isolate computation and data. Adding complexity in this phase (splitting in more sophisticated ways, adding more communications channels and so on, to get better throughput) even is feasible, but still, is orthogonal. The implementation on FPGA of this kernel is made using a software module I wrote to manage the splitting and the update of the matrix at each iteration, while the hardware parts consists of four kernels. At each iteration the software sends to the four kernels the data needed and waits for the computation. Once every kernel has finished, the CPU merges the data and continues to the next iteration.

5.1.2 2mm kernel

The 2mm kernel is a matrix multiplication of the form: $A \times B \times C$. If we look at the equation $D = A \times B$ and at its expanded mathematical representation:

$$\begin{pmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,m} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n,1} & d_{n,2} & \cdots & d_{n,m} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,k} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,k} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k,1} & b_{k,2} & \cdots & b_{k,m} \end{pmatrix} \tag{5.1}$$

A single element in D can be computed as the sum of the product of a single row of A and a single column of B, resulting in the formula:

$$d_{x,y} = \sum_{i=1}^{k} a_{x,i} \times b_{i,y} \tag{5.2}$$

This can be extended to multiple matrices, so if we have $A_1 \times A_2 \times \cdots \times A_n$, with sizes $s_0 \times s_1, s_1 \times s_2, \cdots \times s_{n-1} \times s_n$ the formulae will be:

$$(A_1A_2\cdots A_n)_{i_0,i_n} = \sum_{i_1=1}^{s_1}\sum_{i_2=1}^{s_2}\cdots\sum_{i_{n-1}=1}^{s_{n-1}}(A_1)_{i_0,i_1}(A_2)_{i_1,i_2}\cdots(A_n)_{i_{n-1},i_n} \tag{5.3}$$

Unfortunately, 5.3 can not be used since it would require too many resources, since it requires almost all the entire matrices to compute only one value. On the other side, the two matrices multiplication is simpler than multi matrices product, so it can be easily implemented in a parallel and efficient way without wasting memory and communication time, as I can allocate BRAM only for one row and one column for each element $d_{x,y}$. Since I want to parallelize the computation, I choose to separate the workload on four kernels, each of them computing exactly one quarter or the resulting matrix.

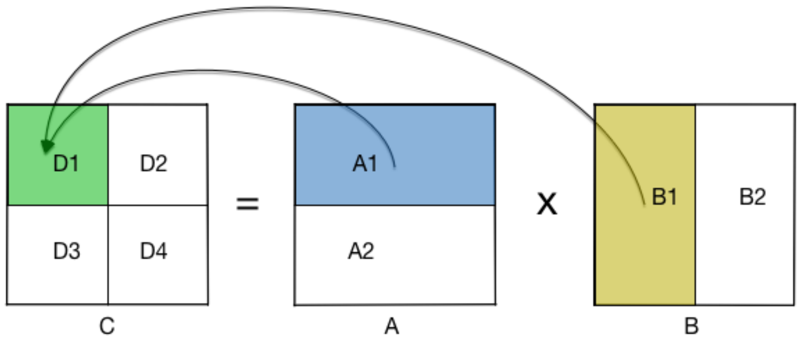


Figure 5.3: Two Matrices Product

In the above figure, I highlight the part of the matrices involved in computation of the first quarter of the resulting matrix. Since I want to split the matrices in four equal parts and given the nature of the matrix product operation, in order to compute one quarter, I need exactly one half of each matrix involved in the operation. In particular, in order to compute the first quarter of C, C1, I need the upper half of A, A1, and the left part of B, B1. For the other three kernels the computation is similar: C2 needs A1 and B2, C3 needs A2 and B1, C4 needs A2 and B2. With this method I am able to split the computation on as many kernels as we want, and achieve higher throughput. Another important consideration to enhance throughput is about how to pass the data through the computational unit, be them software or hardware. In fact, in order to efficiently compute all the operations I need to exploit the data layout. As Jacobi 2-D, in which I split the matrices by horizontal stripe due to the sequentiality of the matrix representation in memory, I need a similar access pattern for both the matrix involved in the product operation. In Figure 5.3 it is clear that the first matrix is accessed by horizontal band while the second is accessed by vertical band. The vertical band imply huge overhead in order to collect the data needed to the computation.

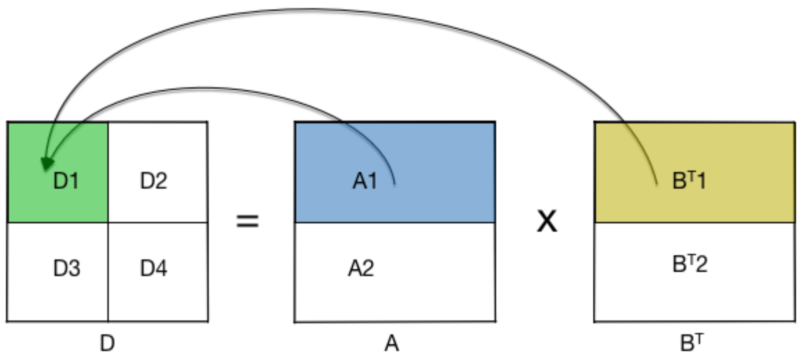


Figure 5.4: Two Matrices Product With the transpose

In Figure 5.4 I represent how a matrix multiplication should be to be efficient regards memory communication. If I use this method it will clearly produce uncorrect results, but if instead of B we choose the transpose of B, B^T, and change the result equation in:

$$d_{x,y} = \sum_{i=1}^{k} a_{x,i} \times b_{y,i} \tag{5.4}$$

I am able to compute the product of two matrices more efficiently as I take in account the data layout.

The best method to accelerate 2mm is then to compute as fast as possible the first product and then compute the second one. This is due to the dependence I spoke about before when showing Equation 5.3.

Pseudocode 31 2mm sequential code

```
for i=0 to NI do
    for i=0 to NJ do
        for i=0 to NK do
            C[i*NJ + j] += A[i*NK + k] * B[k*NJ + j]
        end for
    end for
end for
for i=0 to NI do
    for i=0 to NL do
        for i=0 to NJ do
            E[i*NL + j] += C[i*NJ + k] * D[k*NL + j]
        end for
    end for
end for
```

Looking at the sequential code above it, is clear that is not possible to perform the multi-matrices computation without consuming a huge amount of resources, a price that is not affordable. So, my hardware implementation is composed of only a product of two matrices. The software part, instead, splits the computation and sends the data to the kernels as described above, and after the first product is complete, I start another product between the resultant matrix and the last matrix.

5.1.3 3mm kernel

The 3mm kernel is a three multiplication matrix of the form: $A \times B \times C \times D$. This kernel is similar to 2mm. It differs only in the number of matrices multiplied. Since there are four matrices to multiply, and knowing that the product of matrices is associative, we can rearrange the kernels in a very parallel way. In fact, thanks to the associativity, we are able to compute $A \times B$ and $C \times D$ without compromising the correctness of the computation. After the two products are computed, we can multiply the resultant matrices.

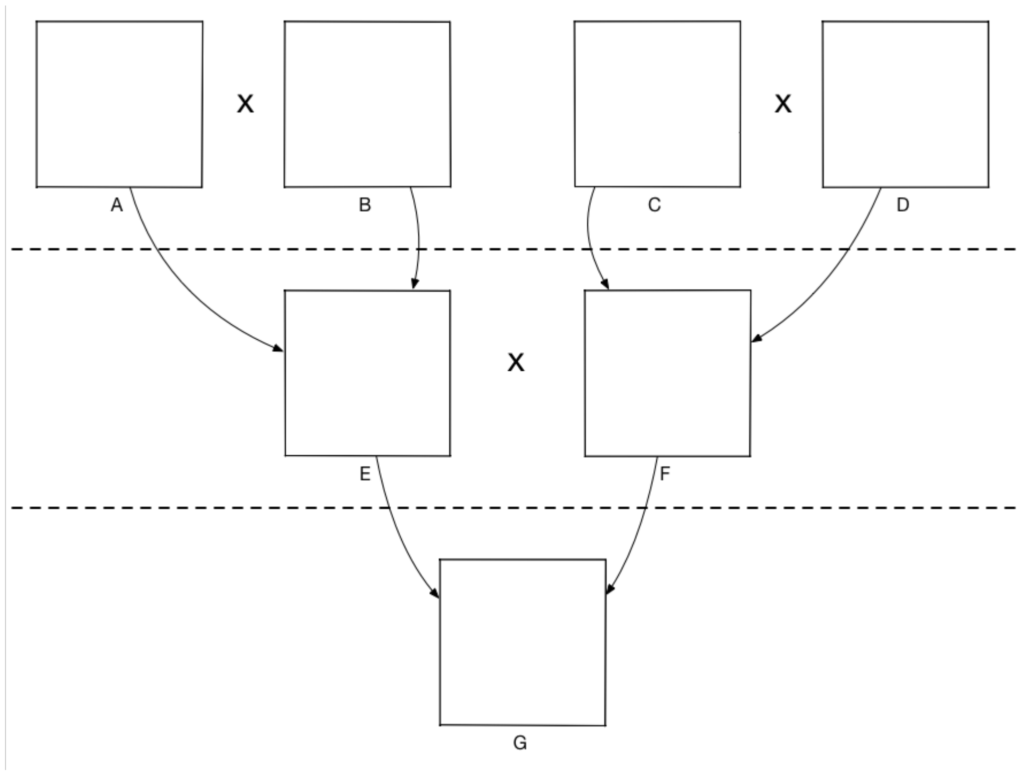


Figure 5.5: 3mm computation scheme

For the parallelization part we used the same steps described in the 2mm kernel. The only difference is that I synthesize two matrix multipliers instead of one. This schema allows us to produce the two resultant matrices at the same time (matrices E and F from 5.5). Looking at the below sequential code below, similarities between 3mm and 2mm are clear. In fact, the same considerations explained on matrix multiplication are true in this case, too. The implementation on FPGA is also similar to the one for the previous kernel. My hardware implementation is composed of two[1] computational units for two matrix products. The software part splits the computation and sends the data to the hardware cores: after both matrices (matrices E and F from 5.5) are computed, the last product is triggered.

[1]Every computation unit is built up using 4 different cores

Pseudocode 32 3mm sequential code

```
for i=0 to NI do
    for i=0 to NJ do
        for i=0 to NK do
            E[i][j] += A[i][k] * B[k][j]
        end for
    end for
end for

for i=0 to NJ do
    for i=0 to NL do
        for i=0 to NM do
            F[i][j] += C[i][k] * D[k][j]
        end for
    end for
end for

for i=0 to NI do
    for i=0 to NL do
        for i=0 to NJ do
            G[i][j] += E[i][k] * F[k][j]
        end for
    end for
end for
```

5.1.4 2-D convolution kernel

The 2-D convolution kernel produces the convolution of an input matrix and some convolution matrix. This type of kernels is mostly used in image processing, like sharpening or edge detection, or more complex and relevant algorithms like convolutional neural networks.

Even if the matrix convolution is a product of matrices, is not the usual multiplication operation between them. In fact, is not a row by column product, but is an element by element operation. Let A be the input matrix, B the convolution matrix, C the resultant matrix, x and y the index of row and column of the element we are computing, respectively, and with the assumption that each matrix has zero as the first index of row and column, the mathematical relation (excluding the border element that are not computed) is:

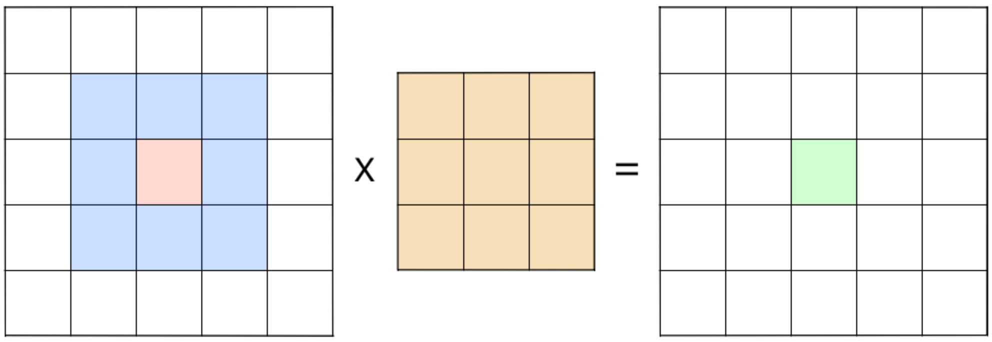


Figure 5.6: 2-D Convolution Computation

$$c_{x,y} = \sum_{i=-1}^{+1} \sum_{j=-1}^{+1} a_{x+i,y+j} \times b_{1+i,1+j} \tag{5.5}$$

This convolution is a nine points operation. It means it takes nine elements of the input matrix in order to compute one elements of the output matrix. The snippet below shows an example of how a convolution code looks like. In this example (taken from Polybench [107]) the convolution matrix is composed by a single value instead of a group inside a matrix, but the meaning is the same.

Pseudocode 33 2D Convolution sequential code

```
1: for i=1 to NI-1 do
2:     for i=1 to NJ-1 do
3:         B[i][j] = c11 * A[i - 1][j - 1] + c12 * A[i + 0][j - 1] + c13 * A[i + 1][j - 1] + c21
           * A[i - 1][j + 0] + c22 * A[i + 0][j + 0] + c23 * A[i + 1][j + 0] + c31 * A[i - 1][j
           + 1] + c32 * A[i + 0][j + 1] + c33 * A[i + 1][j + 1]
4:     end for
5: end for
```

Even if the meaning of the operation is different, parallelizing this kernel involves almost the same operation done with Jacobi 2-D stencil (see 5.1.1). In fact, we have the same border effect, so splitting the workload on multiple kernels requires we pass the hardware units all the data needed, and in particular we need to pass an additional row for the first and the last kernels, and additional two rows for each kernels in between. Since this computation is not iterative, once the hardware unit has computed all the elements assigned, the computation is finished. So, the software part only operates the splitting of the data, sends them to and retrieves them from the hardware components, while the hardware consists of four cores.

5.1.5 BiCG Sub-kernel

In numerical linear algebra, the bi-conjugate gradient stabilized method (often abbreviated as BiCGSTAB) is an iterative method developed by H. A. Van Der Vorst for the numerical solution of non-symmetric linear systems [108]. Inside the Polybench suite is present the sub-kernel for computing the two direction vectors.

Pseudocode 34 BiCG Sub-kernel sequential code

```
 1: for i=0 to NI do
 2:    s[i] = 0
 3: end for

 4: for i=0 to NI do
 5:    q[i] = 0
 6:    for i=0 to NJ do
 7:       s[j] = s[j] + r[i] * A[i][j]
 8:       q[i] = q[i] + A[i][j] * p[j]
 9:    end for
10: end for
```

Looking at the above code (Algorithm 34) it is clearly that the two vectors can be computed completely independently from each other. Analyzing the statements on line 7 and 8 in the code, they look similar, yet different. This two statements implement the same kind of operation: vector-matrix multiplication. Broadly speaking, the BiCG Sub-kernel can be viewed as a two matrix-vector product run in parallel. Being a particular case of matrix-matrix product, it can be extremely parallelized.

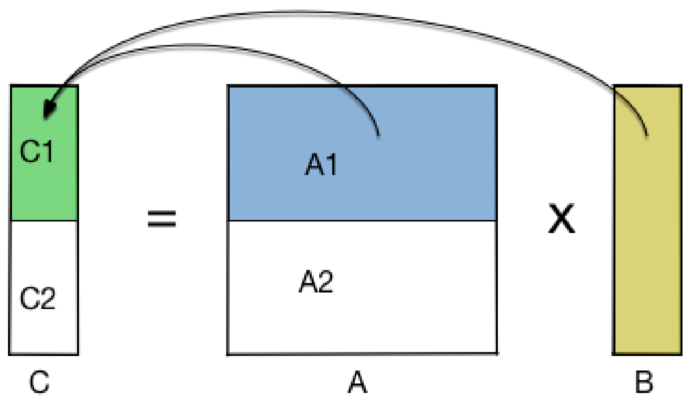


Figure 5.7:
Schema Parallel Hardware Design of vector-matrix multiplication, where the vector is a column vector

In Figure 5.7 is shown how a matrix-vector product can be parallelized, given that the vector can be synthesized with the resources available on the FPGA. This is exactly how the statement on line 8 of algorithm 34 can be computed in parallel. Since one is a column vector (i.e. q) and the other is a row vector (i.e. s) they have to be computed in two different ways. In the case of the computation on line 7 I have to choose one of two possible solutions. As I can see the algorithm is written to exploit the same matrix (i.e. A) for both computation. So I have to choose between:

- Compute the row vector utilizing the transpose of the input matrix
- Compute the multiple partial row vectors and sum them at the end

In the first case I change the algorithm to compute the row vector s as we compute the column vector q. But we have to pay the time needed to compute the transpose. In the second one I create more row vectors, each one will contain the partial sum, and then sum them together to obtain the final values (as we can see in figure 5.8).

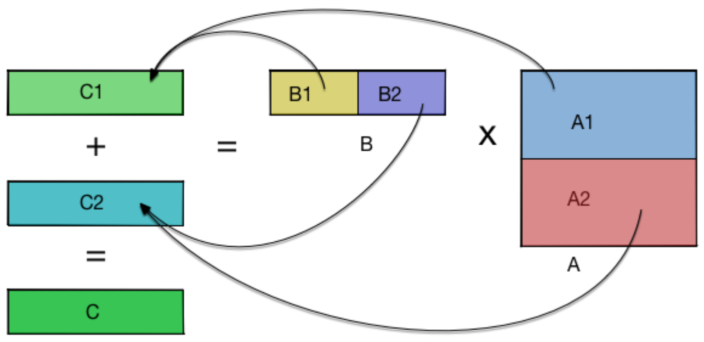


Figure 5.8:
Schema Parallel Hardware Design
of vector-matrix multiplication,
where the vector is a row vector

In this case I have to pay a little error in computation due to the additional final sum needed to compute the resulting vector. As the generation of the transpose in this case does not give any advantage, but actually slows down the entire process, I choose to opt for the second approach. My hardware solution is composed of eight kernels, four for each vector-matrix product. The software part, as the previous kernel, splits the data and passes them to the hardware. At the end of the computation I need to reorder the partial vectors in to the final ones. This is also the final work of the software part.

5.2 Implementation Results

For each kernel described in the previous section, I had implemented three different versions:

- Simple HLS
- Split-Down
- Split-Down, with directives and memory optimization enabled
- Theoretical best

Simple HLS means that I took the kernel code, and I simply added the interfaces needed to communicate between the main memory and FPGA to implement a functional hardware design.

Split-Down means that we implemented a parallel version of the kernel without relying on hardware optimization.

Split-Down, with directives and memory optimization enabled means that I implemented a parallel version of the kernel enhancing BRAMs usage and setting the High Level Synthesis (HLS) tools to use dataflow and pipelining directives to optimize the generated hardware circuits.

Theoretical best means the best possible acceleration with only one Zedboard, increasing the number of parallel cores synthesized. I can assume a linear increment in performance rising hardware resources.

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

Table 5.1: Simple HLS

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

Table 5.2: Split-Down

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

Table 5.3: Split-Down, with directives and memory optimization enabled

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

Table 5.4: Theoretical best

As expected, the generation of parallelized hardware gives better results than plain conversion of sequential code. Since we are targeting Xilinx FPGA's, given the availability of the Zedboard, the *problem dimension* fills the FPGA's BRAM. While this precious resource is almost used completely, all other resources like LUTs or DSPs were almost unused, a clear sign of bad usage of board resources.

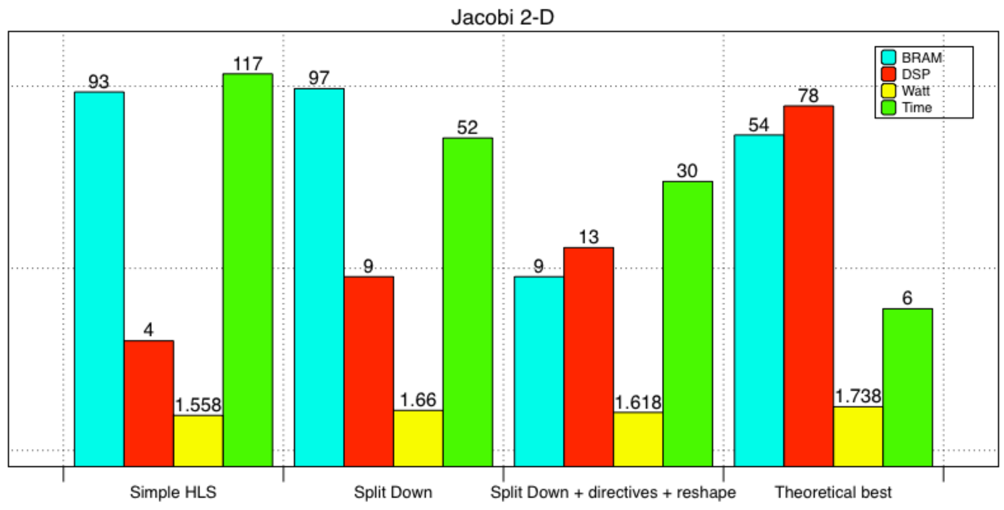


Figure 5.9: Jacobi 2-D resource chart

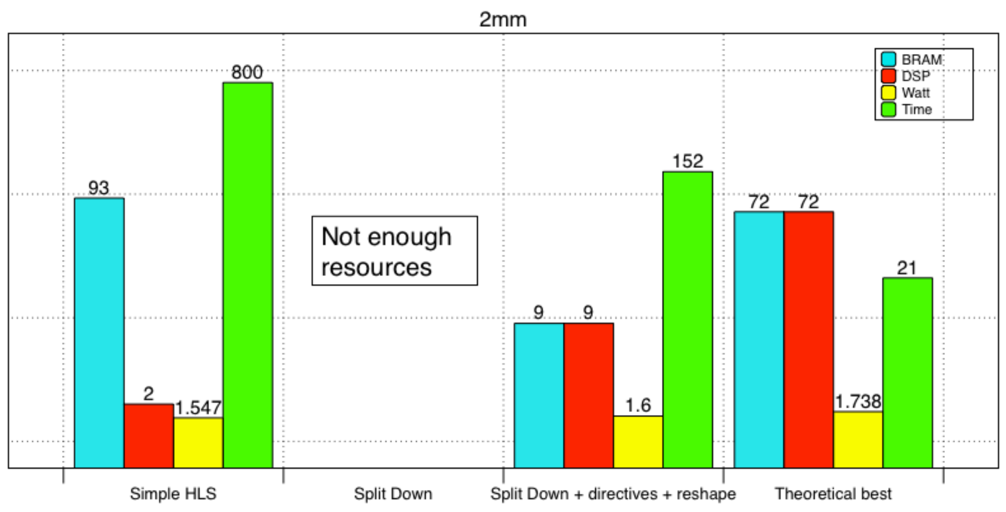


Figure 5.10: 2mm resource chart

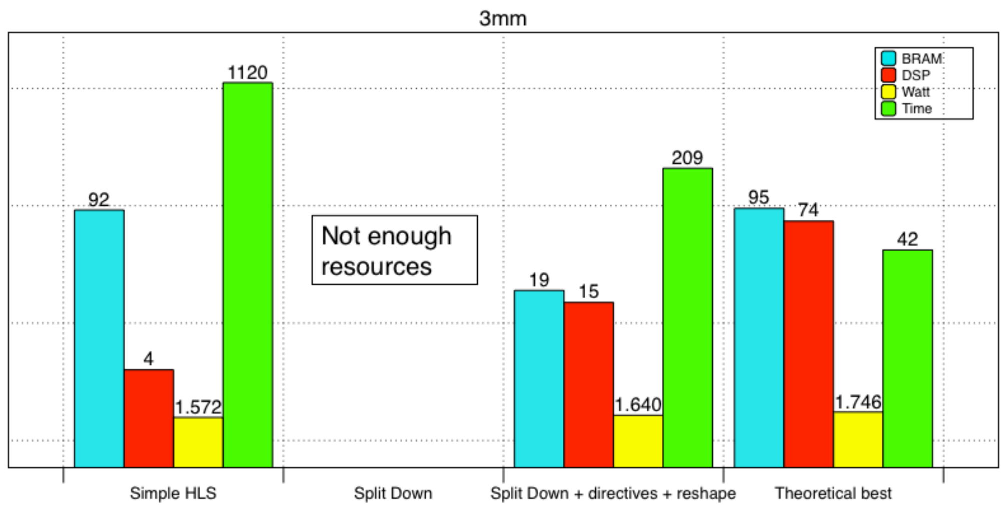


Figure 5.11: 3mm resource chart

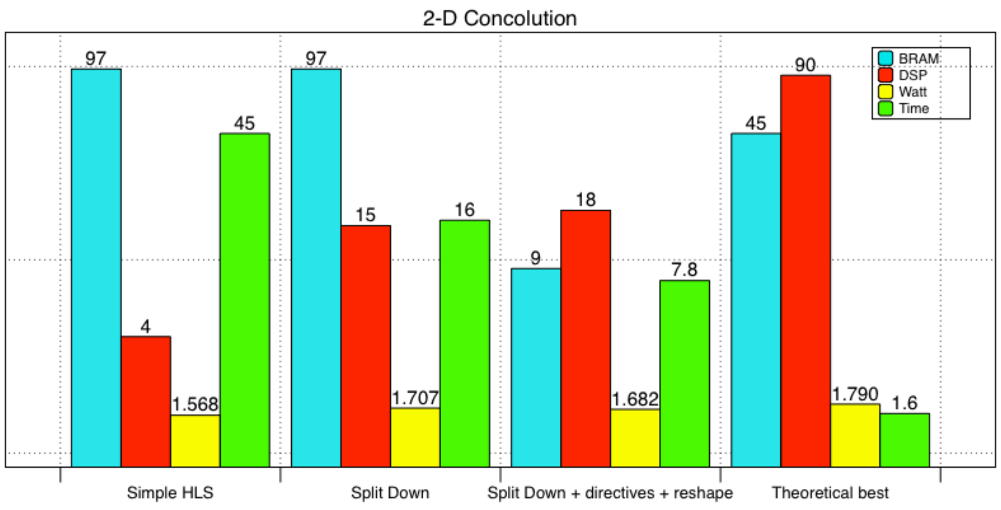


Figure 5.12: 2-D Concolution resource chart

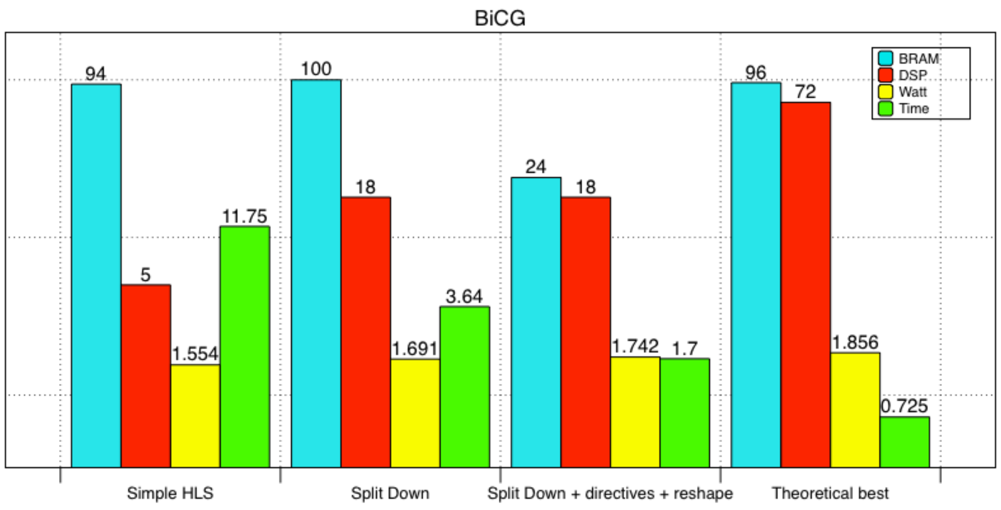


Figure 5.13: BiCG resource chart

So I created the parallel version with four cores for each kernel. Since I need to pass data to the computational unit, I had to determine the correct method. I opted to synthesize one DMA controller for each core in order to send and receive data in a completely asynchronous way. Other implementations can be made, but I chose this solution, as it delivers better speed up since each DMA controller can transfer data independently from each other.

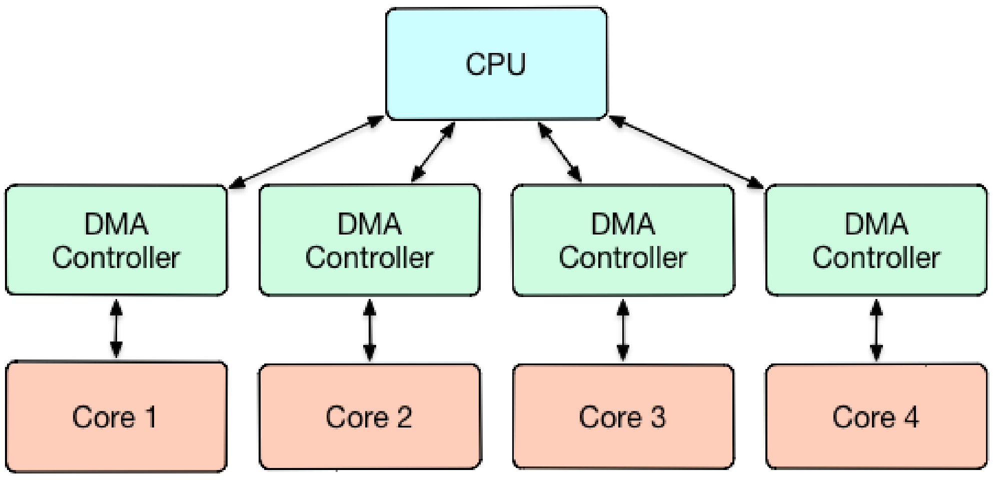
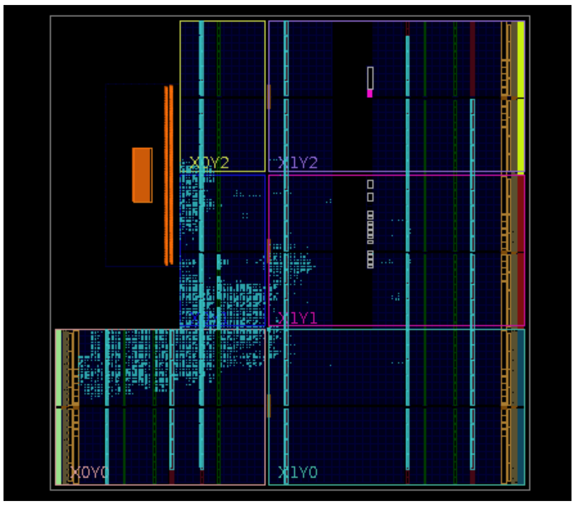
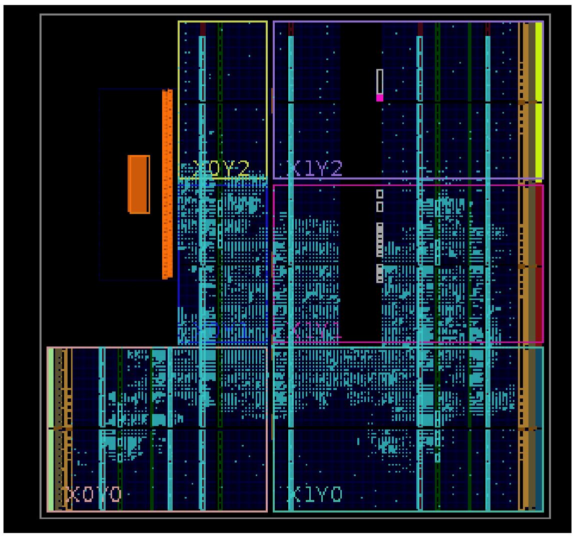


Figure 5.14: Schema Parallel Hardware Design

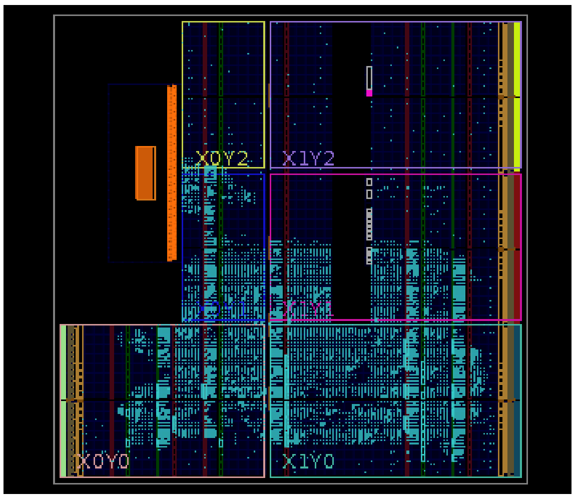
The real comparison must be done between the two parallel versions of the tested kernels. As we can see in 5.2, not all the kernels could be synthesized on the target FPGA, as the usage of the BRAM could be excessive. As we noted above, the 2mm and 3mm kernels are not available in their simpler split-down version as the resources requested outpace by far the resources available on the FPGA. So in this case we cannot make a true comparison, but based on the implementation of the others kernels, we can show that generating a circuit using data flow and pipelining directives delivers a huge speed up. In the "simple HLS" and "Split-Down" versions these directives have not been used since the limited BRAM resources on the Zedboard reached its physical limits. We had to rearrange the kernel code to stay inside its limits creating a smaller "cache" inside each core in order to use them.



(a) Floorplan Simple HLS



(b) Floorplan Split Down HLS



(c) Floorplan Split Down plus reshape memory and directives HLS

Figure 5.15: Area used on Jacobi 2-D

Since I treat hardware cores in a way similar to OS's threads, software and hardware implementations are very similar. Thus we can reduce the problem of design hardware to the one of creating a threaded software. So if we are able to manipulate the original code and create a parallel version, then we can generate a correct architecture that is capable of do the same process of software but in hardware. Clearly, the software implementation will not be written as normal threaded code, but since our problem is describing the circuits behavior, we should consider more hardware-friendly implementation of the original code.

6

Conclusions and Future work

6.1 Conclusions

In the current work, I explained how Polyhedral Model (PM) can be used to restructure the code to achieve better parallelization obtaining as a final result an increase of the efficiency of hardware circuits. The methodology proposed faces the problem of creating an architecture suitable for the problem. In my experimental test I show a slight increase in power consumption in the more parallel architecture respect to straight implementation, but is largely compensated by a speed up ranging from 3x to 7x in total calculation time. In fact, the worst power consumption increase is of the order of 6%(about 100mW), a small price spent compared to the huge gain in throughput which directly translates into better power efficiency. You should note that the real price comes from the total energy consumed as it comes from the product of total time of computation multiplied by Watts spent. As I explained in the previous chapter this methodology can be considered valid only for algorithm intrinsically parallels. Also, I need to limit the expressivity of the language to be pure: implementing synchronized access to shared data not only will result in bottlenecks, but also a lot of effort should be put to design this features. However, as the focus, is on scientific algorithms, this is not a real issue, as most of those algorithms already are expressed in this form.

6.2 Future works

The first goal is to implement the final toolchain. The second goal is to tighten the integration between task carried on by difference researchers, like for example [78, 79, 80]. The third future goal will be to prove the feasibility of the multi-Field Programmable Gate Array (FPGA) solution in order to implement memory intensive algorithms. I will conduct extensive tests on other various computational kernels split on multiple FPGA retrieving statistics and performance counts will demonstrate how the performance scales out. In Low-Level Virtual Machine (LLVM) related area, we already use Clang (via Chunky Loop ANalyzer (CLAN)) to translate code into PM. Till now we are using source-to-source transformation to create C file to feed Xilinx tools. Following the typical LLVM schema (front-end, IR, back-end) we could implements a different tool, that exploit directly the intermediate representation to generate the Register-Transfer Level (RTL) of the circuits: in other words we could implement a typical LLVM back-end. The first step in this direction will be analyze and profile what has been already developed by LegUp project.

Bibliography

[1] Gordon E Moore and Life Fellow. Cramming More Components onto Integrated Circuits. 86(1):82–85, 1998.

[2] index @ www.energy.gov.

[3] index @ science.energy.gov.

[4] The Opportunities and Challenges of Exascale Computing Fall 2010 Report on Exascale Computing.

[5] Avinash Sodani and D Ph. Race to Exascale : Opportunities and Challenges Intel Corporation.

[6] Dimitri Kusnezov, Senior Advisor, and U S Doe. DOE Exascale Initiative. pages 1–12, 2013.

[7] Michael B Taylor. Is Dark Silicon Useful ? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse U " liza " on Wall :. 2005.

[8] index @ www.exascale-computing.eu.

[9] physx @ www.geforce.com.

[10] high-performance-xeon-phi-coprocessor-brief @ www.intel.com.

[11] Xilinx. www.xilinx.com.

[12] sdaccel @ www.xilinx.com.

[13] Jason Cong, Muhuan Huang, and Yi Zou. Accelerating Fluid Registration Algorithm on Multi-FPGA Platforms. *2011 21st International Conference on Field Programmable Logic and Applications*, pages 50–57, September 2011.

[14] Mohammad H Al-towaiq. Parallel Implementation of the Gauss-Seidel Algorithm on k -Ary n -Cube Machine. 2013(January):177–182, 2013.

[15] Juanjo Noguera and Fernando Martinez Vallina. Zynq-7000 All Programmable SoC Accelerator for Floating-Point Matrix Multiplication using Vivado HLS. 1170, 2013.

[16] Vincenzo Rana, Alessandro A Nacci, Ivan Beretta, Marco D Santambrogio, David Atienza, and Donatella Sciuto. Design Methods for Parallel Hardware Implementation of Multimedia Iterative Algorithms. (c):1–6, 2011.

[17] cuda_home_new @ www.nvidia.com.

[18] mantle @ www.amd.com.

[19] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. *2008 Symposium on Application Specific Processors*, pages 101–107, June 2008.

[20] Altera Corporation. Radar Processing : FPGAs or GPUs ? (May), 2013.

[21] Kuen Hung Tsoi and Wayne Luk. Axel : A Heterogeneous Cluster with FPGAs and GPUs. 2010.

[22] Blue Book. High-Level Synthesis.

[23] P. Coussy, D.D. Gajski, M. Meredith, and a. Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, July 2009.

[24] index @ www.altera.com.

[25] Dan Gajski, U C Irvine, and Irvine Ca. What Input-Language is the Best Choice for High Level Synthesis (HLS)? pages 857–858, 2010.

[26] Matthijs Kooijman, Christiaan Baaij, and Jan Kuper. From Haskell To Hardware.

[27] Edward A Lee. Heterogeneous Concurrent Modeling and Design in Java (Volume 1 : Introduction to Ptolemy II). 1, 2008.

[28] Edward A Lee and Stephen Neuendorffer. Heterogeneous Concurrent Modeling and Design in Java (Volume 2 : Ptolemy II Software Architecture). 2, 2008.

[29] Christophe Lucarz and Marco Mattavelli. Dataflow / Actor-Oriented language for the design of complex signal processing systems. (Dasip), 2008.

[30] Christophe Lucarz, Marco Mattavelli, and Julien Dubois. A Platform for the Development and the Validation of HW IP Components Starting from Reference Software Specifications. *EURASIP Journal on Embedded Systems*, 2008(1):685139, 2008.

[31] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanovi. Chisel : Constructing Hardware in a Scala Embedded Language.

[32] Richard Thavot, Romuald Mosqueron, Julien Dubois, and Marco Mattavelli. Hardware synthesis of complex standard interfaces using CAL dataflow descriptions.

[33] Johan Eker. Specification of the C AL actor language. 2003.

[34] Christian Feichtinger, Johannes Habich, Harald Köstler, Georg Hager, Ulrich Rüde, and Gerhard Wellein. A flexible Patch-based lattice Boltzmann parallelization approach for heterogeneous GPU–CPU clusters. *Parallel Computing*, 37(9):536–549, September 2011.

[35] Alexandru Fiodorov. *Improving Energy Efficiency with Special-Purpose Accelerators*. PhD thesis, Norwegian University of Science and Technology, 2013.

[36] Richard Membarth, Frank Hannig, Jurgen Teich, and Harald Kostler. Towards Domain-Specific Computing for Stencil Codes in HPC. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1133–1138, November 2012.

[37] Dmitry Nadezhkin. *Parallelizing Dynamic Sequential Programs using Polyhedral Process Networks*.

[38] Alejandro Fernández Suárez. Domain Specific Languages for High Performance Computing A Framework for Heterogeneous Architectures. pages 2012–2013, 2013.

[39] S. van Haastregt and B. Kienhuis. Automated synthesis of streaming C applications to process networks in hardware. *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 890–893, April 2009.

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

[41] Daniel L Slotnick, W Carl Borck, and Robert C Mcreynolds. The solomon computer*. 30(December):97–107, 1962.

[42] Sjoerd Meijer. *Transformations for Polyhedral Process Networks*.

[43] Charles Severans and Kevin Dowd. Understanding Parallelism - Loop-Carried Dependencies.

[44] Christian Lengauer. Loop Parallelization in the Polytope Model. pages 1–19.

[45] Paul Feautrier. *The Polytope Model : Past , Present , Future What is a Model ?* 2009.

[46] Amy W Lim and Monica S Lam. Maximizing Parallelism and Minimizing Synchronization with A ne Transforms 2 Forms of Parallelism.

[47] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations. pages 9–18.

[48] Wei Zuo, Peng Li, Deming Chen, Louis-Noel Pouchet, and Jason Cong. Improving polyhedral code generation for high-level synthesis. *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, September 2013.

[49] C. Bastoul. Code generation in the polyhedral model is easier than you think. *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, 2004.

[50] Programming Group. May 1, 2012 14:53 WSPC/INSTRUCTION FILE paper. 2012.

[51] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: A Tool for Improved Derivation of Process Networks. *EURASIP Journal on Embedded Systems*, 2007:1–13, 2007.

[52] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset. High-Level Synthesis of Loops Using the Polyhedral Model The MMAlpha Software.

[53] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC'08/ETAPS'08 Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, pages 132–146, 2008.

[54] Mohamed-walid Benabderrahmane and Albert Cohen. The Polyhedral Model Is More Widely Applicable Than You Think.

[55] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism.

[56] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-Model Guided Loop-Nest Auto-Vectorization. *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337, September 2009.

[57] Anders Nilsson and Karl-erik Å rzén. Static Analysis and Transformation of Dataflow Multimedia Applications. (November), 2012.

[58] Tomofumi Yuki. AlphaZ and the Polyhedral Equational Model.

[59] Donatella Sciuto Advisor and Marco D Santambrogio Advisor. DATA LEVEL PARALLELISM WITH POLYHEDRAL PROCESS. 2014.

[60] Locality Analysis. Dependence Analysis and Loop Transformations.

[61] Andreas Simbürger, Sven Apel, Armin Größ linger, and Christian Lengauer. The Potential of Polyhedral Optimization The Potential of Polyhedral Optimization. (February), 2013.

[62] Mary Hall. Compiler-Based Autotuning Technology Lecture 3 : A Closer Look at Polyhedral Compiler Technology Polyhedral Compiler Technology. 2011.

[63] Benoit Pradelle, Alain Ketterlin, and Philippe Clauss. Polyhedral parallelization of binary code. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–21, January 2012.

[64] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral Code Generation in the Real World.

[65] Cédric Bastoul, Albert Cohen, Sylvain Girbal, and Saurabh Sharma. Putting Polyhedral Loop Transformations to Work.

[66] Arnamoy Bhattacharyya and José Nelson Amaral. Automatic speculative parallelization of loops using polyhedral dependence analysis. *Proceedings of the First International Workshop on Code OptimiSation for MultI and many Cores - COSMIC '13*, pages 1–9, 2013.

[67] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing Transformations for Locality Enhancement of Imperfectly-Nested Loop Nests. 29(5):493–544, 2001.

[68] A Loop Generator and For Scanning. C´ edric Bastoul. 2007.

[69] The Polyhedral and Compiler Collection. PoCC. 2013.

[70] Martin Griebl and Christian Lengauer. The Loop Parallelizer LooPo 1 Why LooPo ? 2 Theoretical Background.

[71] Armin Gr. The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation. 2009.

[72] Armin Größ linger. Precise Management of Scratchpad Memories for Localising Array Accesses in Scientific Codes. *CC '09 Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, pages 236–250, 2009.

[73] Louis-noël Pouchet, Cédric Bastoul, and Albert Cohen. LetSee : the LEgal Transformation SpacE Explorator.

[74] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*, page 101, 2008.

[75] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. Data reuse analysis technique for software-controlled memory hierarchies. *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, pages 202–207, 2004.

[76] Paul Feautrier. Data ow Analysis of Array and Scalar References. (September):1–37, 1991.

[77] Christian Pilato, Politecnico Milano, Politecnico Milano, and Politecnico Milano. A Design Methodology to Implement Memory Accesses in High-Level Synthesis. pages 49–58.

[78] J. Cong. Behavior and communication co-optimization for systems with sequential communication media. *2006 43rd ACM/IEEE Design Automation Conference*, pages 675–678, 2006.

[79] Jason Cong, Stephen Neuendorffer, Juanjo Noguera, and Kees Vissers. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[80] Jason Cong, Peng Zhang, and Yi Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, page 1233, 2012.

[81] Amy W Lim, Gerald I Cheonp, and Monica S Lam. An Affine Partitioning Algorithm to Maximize Minimize Communication Parallelism and. pages 228–237, 1999.

[82] Fabien Quiller E. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, September 2000.

[83] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable Domain-Specific Computing. *IEEE Design & Test of Computers*, 28(2):6–15, March 2011.

[84] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '13*, page 29, 2013.

[85] Sven Verdoolaege. Polyhedral Process Networks. pages 1–35.

[86] H Nikolov, M Thompson, T Stefanov, A Pimentel, and Application-based Systems Real-time. Daedalus : Toward Composable Multimedia MP-SoC Design. pages 574–579.

[87] Hristo Nikolov, Student Member, Todor Stefanov, and Ed Deprettere. Systematic and Automated Multiprocessor System. 27(3):542–555, 2008.

[88] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Multi-processor system design with ESPAM. *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis - CODES+ISSS '06*, page 211, 2006.

[89] Marco Lattuada, Fabrizio Ferrandi, and Milano Dipartimento. Performance Modeling of Embedded Applications with Zero Architectural Knowledge. pages 277–286, 2010.

[90] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp : High-Level Synthesis for FPGA-Based Processor / Accelerator Systems. pages 7–10.

[91] Vito Giovanni Castellana and Politecnico Milano. An Automated Flow for the High Level Synthesis of Coarse Grained Parallel Applications. pages 294–301, 2013.

[92] Silvia Lovergine and Fabrizio Ferrandi. Harnessing Adaptivity Analysis for the Automatic Design of Efficient Embedded and HPC Systems. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 2298–2301, May 2013.

[93] Roberto Cordone, Milano Dti, and Marco D Santambrogio. Using Speculative Computation and Parallelizing techniques to improve Scheduling of Control based Designs.

[94] Vito Giovanni Castellana, Fabrizio Ferrandi, and Milano Dipartimento. Scheduling Independent Liveness Analysis for Register Binding in High Level Synthesis. 2013.

[95] http://clang.llvm.org/.

[96] A Specification. C´ edric Bastoul. 2014.

[97] Todor Stefanov. *Converting weakly dynamic programs to equivalent process network specifications*. Phd thesis, Leiden University, 2004.

[98] Jörn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing Hardware from Dataflow Programs. *Journal of Signal Processing Systems*, 63(2):241–249, July 2009.

[99] Shuvra S Bhattacharyya, Gordon Brebner, and Johan Eker. How to make stream processing more mainstream. pages 2–4.

[100] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cvclo-Static Dataflow 4. 44(2), 1996.

[101] Eunjung Park, Louis-Noel Pouche, John Cavazos, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 119–129, April 2011.

[102] A X I Reference, Vivado Axi, and Reference Guide. Vivado Design. 1037:1–143, 2014.

[103] Design Suite. AXI4-Stream Infrastructure IP Suite Table of Contents. 2013.

[104] High-level Synthesis. Vivado Design Suite Tutorial. 871, 2013.

[105] High-level Synthesis. Vivado Design Suite User Guide. 902, 2013.

[106] Xilinx. Xilinx Vivado Design Suite Tcl Command Reference Guide (UG835). 835, 2012.

[107] Louis-Noël Pouchet. PolyBench/C the Polyhedral Benchmark suite.

[108] H A van der Vorst. BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, 1992.

December 2, 2014

Document typeset with LaTeX