PolyFPGA:
A tool to automatically accelerate iterative stencil loops

Relatore: Prof. Marco Domenico SANTAMBROGIO
Correlatore: Dott. Ing. Riccardo Cattaneo

Tesi di Laurea di:
Giulio Stramondo
Matricola n. 800632

Anno Accademico 2014–2015
Contents

1 Introduction 3
   1.1 High Performance Computing: a Perspective 3
   1.2 Milestones in High Performance Computing 5
   1.3 Exascale-class High Performance Systems 5
      1.3.1 Field Programmable Gate Array (FPGA) 7
   1.4 Structure of the Thesis 10

2 State Of the Art 12
   2.1 FPGA Overview 12
   2.2 Pre-HLS design flow 14
      2.2.1 High Definition Synthesis tools 16
   2.3 High Level Synthesis 19
      2.3.1 High Level Synthesis tools 20
   2.4 Electronic System-Level design 24
      2.4.1 Electronic System-Level tools 26
   2.5 PolyFPGA 31

3 Problem Statement and Contribution 33
   3.1 Problem Statement 33
   3.2 Contribution 35

4 Polyhedral Model 36
5.3.3 Synthesis ................................................. 59

6 PolyFPGA Workflow ................................. 60
6.1 Introduction ........................................... 61
6.2 Input file ............................................. 62
6.3 Parsing Module ..................................... 65
6.4 Evaluation Module ................................. 67
6.5 Streaming Oriented Graph Module ............... 67
6.6 SOAG Module ..................................... 70
  6.6.1 Computing the Stencil Array Size .......... 71
  6.6.2 Computing the FIFO sizes ................. 73
  6.6.3 Ordering the filters ....................... 76
6.7 ISCC Interface ..................................... 77
6.8 HLS Generator Module ........................... 78
6.9 SST Packager Module ............................. 79
6.10 Bitstream Generator Module .................... 79
6.11 Firmware Generation ............................. 80
6.12 Conclusions ..................................... 81

7 Experimental Results ......................... 82
7.1 Experimental Setup .............................. 82
7.2 Command Line Interface ....................... 83
7.3 Test Input Codes .................................. 83
  7.3.1 Jacobi 1D ..................................... 83
  7.3.2 Jacobi 2D ..................................... 84
  7.3.3 Heat Equation .............................. 84
7.4 Test Cases ........................................ 85
  7.4.1 Jacobi 1D on Virtex-7 ....................... 85
  7.4.2 Jacobi 2D on Virtex-7 ....................... 88
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.3 Heat Equation 3D on Virtex-7</td>
<td>91</td>
</tr>
<tr>
<td>8  Conclusions and Future Works</td>
<td>94</td>
</tr>
<tr>
<td>Bibliography</td>
<td>103</td>
</tr>
</tbody>
</table>
List of Listing

2.1 VHDL example ........................................... 14
4.1 Loop Nest .................................................. 39
4.2 Jacobi 2D 10x10 ........................................... 49
6.1 Input code example ........................................ 63
6.2 Jacobi 2D kernel ........................................... 66
6.3 Jacobi 2D 10x10 ........................................... 71
7.1 Jacobi 1D .................................................... 84
7.2 Jacobi 2D .................................................... 84
7.3 Heat Equation 3D .......................................... 85
7.4 Jacobi 1D PolyFPGA command .......................... 85
List of Figures

1.1 Graph showing the cost of a 1 Million System Gate FPGA produced by Xilinx over the years [1] ................................. 8

2.1 FPGA Market, source: EETimes ....................................... 13
2.2 RTL representation of a digital circuit ................................. 14
2.3 Electronic system design flow .......................................... 25
2.4 Daedalus design flow .................................................... 27
2.5 System-on-chip design flow ............................................ 28
2.6 Example of SDFG Model ................................................ 30
2.7 MAMPSx Architectural Template ..................................... 30

4.1 Data Domain ............................................................... 49
4.2 5-point, 2D stencil pattern ............................................... 50

5.1 Complete architecture obtained queuing SST modules ............ 53
5.2 SST module ............................................................... 54
5.3 Methodology ............................................................... 56

6.1 PolyFPGA workflow ..................................................... 63
6.2 Data Dependency Graph ............................................... 67
6.3 Data Dependency Graph after edge removal algorithm ........ 68
6.4 Streaming Oriented Graph with copy dependencies ............ 69
6.5 Streaming Oriented Graph ............................................... 70
LIST OF FIGURES

6.6 Iteration Domain ............................................. 72
6.7 Iteration Domain ............................................. 72
6.8 Computation of Array Size .............................. 73
6.9 Obtaining filter’s index vector: A[i][j + 1] .......... 75
6.10 Filter ordering: access A[i][j - 1] ................... 77
6.11 Filter ordering: access A[i][j - 1] ................... 77

7.1 Performances Jacobi 1D .................................. 86
7.2 Resources Jacobi 1D ....................................... 87
7.3 Power Efficiency Jacobi 1D .............................. 87
7.4 Performance Jacobi 2D ................................... 88
7.5 Comparison Jacobi 2D ................................... 89
7.6 Resources Jacobi 2D ....................................... 90
7.7 Power Efficiency Jacobi 2D .............................. 90
7.8 Performances Heat 3D ................................... 91
7.9 Resources Heat 3D ....................................... 92
7.10 Power Efficiency Heat 3D .............................. 92
A wide range of scientific problems can be solved using stencil computations. Many particle interaction, computer vision algorithms as well as methods for solving partial differential equation are based on them.

The aim of this work is to introduce a framework, named PolyFPGA, that allows in a completely automatic way to accelerate this kind of computations. The framework focuses on the Iterative Stencil Loops (ISL), a type of scientific computation, and generates a streaming-based micro-architecture, known in literature as Streaming Stencil Time-step (SST), that can be synthesized on Field Programmable Gate Array. SST-based architectures have shown interesting properties of scalability and data reuse, that permits it to be implemented using a low amount of resources. Until now every hardware accelerator based on SST had to be generated manually; PolyFPGA is the first framework able to perform in a completely automatic way all the operation needed to generate those hardware accelerators. In order to obtain such architecture automatically, the framework leverages some of the mainstream Polyhedral tools, that are smoothly included, and also drives the synthesis using the Xilinx Vivado suite.

The framework extends the methodology to generate SST already present in literature, addressing and solving some concrete problems that arise in the automatic generation of the architecture. It was built in a modular manner in order to allow it to target different FPGA boards and to be easily extended with new features and boards.
Molti problemi di natura scientifica possono essere risolti utilizzando algoritmi basati su stencil computations. Tra questi vi sono sistemi volti a studiare l’interazione tra particelle, algoritmi di computer vision e metodi di risoluzione per equazioni differenziali alle derivate parziali. Per questi motivi questo genere di algoritmi sono largamente studiati negli ambiti di ricerca.

Il lavoro proposto introduce un framework, PolyFPGA, che permette l’accelerazione di algoritmi stencil in modo completamente automatico. Il framework si focalizza sull’analisi degli Iterative Stancil Loops (ISL), un particolare tipo di computazioni scientifiche, e genera una micro-architettura streaming conosciuta in letteratura come Stancil Streaming Time-step (SST) sintetizzabile su FPGA. Le architetture basate sugli SST hanno mostrato interessanti proprietà di scalabilità e riuso di dati, che ne permettono l’implementazione utilizzando un basso numero di risorse. Al fine di ottenere quest’architettura in modo automatico, il framework utilizza alcuni tra i più diffusi tool di analisi poliedrale, che include agevolmente, e guida la sintesi utilizzando il tool Vivado di Xilinx. PolyFPGA estende la metodologia già presente in letteratura, riguardante la generazione automatica degli SST, affrontando e risolvendo vari problemi che sorgono durante la procedura automatica. È costruito in maniera modulare permettendo così utilizzo su diverse schede FPGA e in modo da permettere facilmente l’integrazione di nuove funzioni e schede.
Chapter 1

Introduction

This chapter introduces some fundamental elements of the discourse pertaining this thesis: High Performance Computing (HPC) and the foreseen role of Field Programmable Gate Arrays (FPGAs).

1.1 High Performance Computing: a Perspective

The use of mathematical modeling is diffuse and often necessary to understand the behaviour of natural systems. Its adoption allows to focus the analysis of a real world problem, taking into account only the features correlated to a particular study. An abstract problem can be crafted starting from the original, that can be specified in a rigorous and concise way. Once this representation is obtained it can be used to simulate and predict how the original real world problem reacts to given conditions. A model often allows the researcher to have a trade-off between its complexity and the accuracy that it provides. Therefore, in order to get more precise results, there is the need to consider more aspects of the real world problem.

The main reason why the field of High Performance Computing (HPC) was born is to enable researchers to take advantage of ever more detailed and complex mathematical models. Since this field was born, other scientific and technical
disciplines greatly benefited, and are developing at unprecedented pace.

One of most way used to express the performance of a computer is by stating the Floating Point Operations Per Second (FLOPS) that the computing system can perform. Using such a measuring system is possible to classify the computers in groups with comparable performances, highlighting the evolution pace of HPC. Multiple of the FLOPS are denoted using the following prefixes:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPS</td>
<td>1</td>
</tr>
<tr>
<td>kiloFLOPS</td>
<td>$10^3$</td>
</tr>
<tr>
<td>megaFLOPS</td>
<td>$10^6$</td>
</tr>
<tr>
<td>gigaFLOPS</td>
<td>$10^9$</td>
</tr>
<tr>
<td>teraFLOPS</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>petaFLOPS</td>
<td>$10^{15}$</td>
</tr>
<tr>
<td>exaFLOPS</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>zettaFLOPS</td>
<td>$10^{21}$</td>
</tr>
</tbody>
</table>

As mentioned before, with this measuring system it is possible to classify computers: if a computer has performances of at least one teraFLOP it is said to be part of the class of the terascale computers.

The most widespread way to estimate the FLOPS of a computer is by using the Linpack benchmark. It executes a series of linear algebra kernels on the machine to test and measures the time it takes to compute them, then it gives an estimation on the performance of the computer taking an average on the performances of each kernel. This benchmark is the one currently used to compile the TOP500 list, a list made every year since 1993, containing the fastest 500 supercomputer existing.
1.2 Milestones in High Performance Computing

The first megascale transistors based super computer was the built in 1961 by IBM; it was the IBM 7030, also known as Stretch. This machine started the performance race that continues today with the ever growing hunger for more and more computing power.

The years ranging from 1976 to 1989 are known in the field of super computing as the Cray era. In those years Cray Research developed and produced world’s fastest supercomputers, beginning with the Cray-1, built in 1975 and capable to perform at 160 megaFLOPS. In 1985 the Cray-2 computing system, that had 1.9 gigaFLOPS peak performance, overcame the gigascale barrier. The first terascale computer was built in 1997 by Intel, the Intel ASCI Red. It performed on the Linpack benchmark at 1.068 teraFLOPS. At this time, the fastest super computers are the petascale supercompute; petascale was reach in 2008 by IBM Roadrunner, that achieved a performance of 1.02 petaFLOPS. To date, the most powerful super computer is the Thiane-2, manufactured by NUDT and located in the National Super Computer Center in Guangzhou in China. It has a maximum throughput of 33.86 petaFLOPS and its average power consumption is 24 MW [2] (17 MW of which is consumed by processors, memory and interconnect network, while the rest is used by the cooling system).

1.3 Exascale-class High Performance Systems

The next milestone in the HPC race is the first exascale-class supercomputer. Other than being a significant achievement in computer engineering per sé, being able to build a machine featuring this throughput will allow many fields of research to make gigantic leaps ahead.

The human brain project, for example, is a project that focuses on the study of the capabilities of the brain and tries to fill the gap between technology and biol-
ogy. Having an exascale computer will enable them to have the same computing performances of a human brain. This could bring advancements in both neuroscience, putting the researcher in the position to study and simulate a brain, and in computing technology, trying to harness brain-like features and to use them in new generation processing units [3].

The Nu-FuSE Project is a project financed by the G8 whose goal is to study plasma fusion phenomena, a core scientific problem that needs to be solved before fusion reactors become reality. In order to address such problems, exascale capabilities are required across a range of simulation codes [4].

The European Centre for Medium-Range Weather Forecasts (ECMWF) is an intergovernmental organisation providing medium range weather forecast for 34 European states. They rely for their weather forecast to the Integrated Forecast System (IFS) model that runs on their HPC environment, which is capable of achieving a peak performance of 3.5 petaFLOPS. With their current computers, comprising two identical Cray XC30 systems, they are forced to run simulation with a global grid with a 16 km resolution, but leveraging exascale technology they could reduce this simulation resolution to 2.5 km, therefore delivering much reliable medium-range weather forecasts [5] [6].

The ones above are just few representative examples of the possible applications of an exascale system; many other research fields will indeed benefit from it. Predictions say that these machines will become available by 2020; however, there are multiple issues to cope with before we are effectively able to build one such system.

In fact, suppose that it could be possible to linearly scale the performance of the Thiane-2 system so that it performs at exascale level: if this was the case, that system would require a 30x increase in terms of processing power. However, doing so would also proportionally increase the power consumption of such system (actually, more than linearly, due to network interconnects growing more than
linearly), resulting in a system drawing around 510 MW, the amount of power delivered by the Fort Calhoun nuclear power plant in Nebraska [7]. For this reason, one of the main issues in HPC is to build systems with stringent power efficiency goals in mind.

There are many ways to improve the power efficiency of the current machines. One of them would be to abandon the currently used general purpose architecture and to move towards Coprocessors Dominated Architecture [8], i.e. architectures tailored around specific workloads. There are in fact evidences, as is also shown by some HPC researches [9], that its possible to contemporarily increase throughput while reducing power consumption. This is because for an architecture to be general purpose, it has to allocate silicon area that is not entirely “useful” to carry the computation on, but is nonetheless required in order for it to be flexible enough to adapt to different computations. At the time of writing, the most relevant HPC trend is to use heterogeneous architectures, composed by a CPU and GPU accelerators, in the light of the benefits of heterogeneous, spatial computing architectures.

However, FPGA accelerators are being increasingly seen and employed as core coprocessors due to the high power efficiency, already well demonstrated in multiple application scenarios [10][11][12][13].

1.3.1 Field Programmable Gate Array (FPGA)

FPGA are a special kind of Integrated Circuit (IC) that can be configured by a designer in order to change the hardware function implemented by the circuit. They are composed by many arrays of logic blocks, that implement the most basic functions (like: blocks of RAMs, Digital Signal Processors, Look Up Tables, and the like) which are interconnected via electrically programmable switch boxes. The reconfiguration of the board consists of a rerouting of the arrays interconnection performed by changing the status of the electrically reprogrammable
switches. The logic blocks in an FPGA are mainly based on transistor pairs, basic small gates such as NAND and XOR, multiplexer, Look Up Tables (LUT) and wide fan-in AND-OR structures [14].

This technology, until recently, was applied in embedded application, because the cost and the production technologies weren’t allowing the production of Field Programmable Gate Array (FPGA) suitable for computational intensive tasks. However this scenario is changing rapidly and FPGAs are gaining support also in the HPC area.

Figure 1.1: Graph showing the cost of a 1 Million System Gate FPGA produced by Xilinx over the years [1]

There are many issues that still prevent a wide adoption of FPGAs in HPC.

The most relevant FPGA problem is that the design phase of a specific accelerator for a specific application is a long and complex process that requires most of the time designers specifically trained in hardware and software codesign, a relatively scarce competence to hire. Each accelerator needs to be carefully thought out following a process composed by many phases. To solve the problem, many design tools have been implemented in order to allow less experienced designers to take advantage of the technology and in order to reduce the design production time. When this technology was born the only way to configure the FPGA
boards was to design the circuit using low level specification languages such as VHDL and Verilog. Even if the use of those languages is still used in the field, they are used nowadays only if there is the need to finely tune a particular aspect of a given circuit, more or less in the same way we can find Assembly snippet of codes within C programs, in order to specify the behavior of the most sensitive pieces of code. Clearly, when a computing system is composed by many FPGA boards, and each board has to communicate with the order, as in the case of a multi-FPGA system, the complexity of the designs, and the time to market, rapidly grows. Therefore, especially when designing larger system, the adoption of automation flows is common.

Hardware designers generally develop using multiple levels of abstractions; the design of an architecture begins with an abstract specification at an algorithmic level where the functionality of the desired circuit, as well as its interfaces, are specified. This level of abstraction is still far from the actual hardware, so a more concrete description of the hardware behavior, including the circuit’s timing and the details concerning the clock specifications, are decoupled from the high level design description.

The process of translating an abstract representation into a lower level one is called synthesis. After the definition of an high level design, it is transformed by a process called high level synthesis into a Register-Transfer Level (RTL) specification. At this level the description of the circuit usually happens in some Hardware Description Language (HDL) language such as Verilog or VHDL.

From the RTL level of abstraction, with a process referred to as logic synthesis, the specification is translated into a design implementation, expressed in terms of gates. This specification can at last be used, as an example, to target FPGA by creating the corresponding bitstream.

There are many tools that allow to automatically perform High Level Synthesis (HLS), collectively referred to as HLS tools. They allow the designers to ex-
press the functionality of the hardware using high level programming languages, such as C. Once the hardware specification is written, it is interpreted by the HLS tool that, matching coding patterns with the respective hardware implementation is able to derive a functionally equivalent hardware. Those kind of tools, abstracting away all the complexities derived by using HDL specification, enable to rapidly design more complex systems, therefore reducing the time-to-market of the different projects.

There are many different vendors of HLS tools; among them are Xilinx with Vivado HLS [15], Synopsys’ Synphony C Compiler [16], and Calypto’s Catapult C. Most of the HLS tools were historically conceived to use the FPGA architecture in embedded systems that were composed of a series of Intellectual Property (IP) cores connected among each other using standard interfaces. More recently, though, these tools are becoming employed as core components of more complex design flows where the goal is no more an embedded system but a larger scale computing system. The proposed work fits in this line of research.

I present a framework, PolyFPGA, which is able to generate, in a completely automated manner, the design of the entire system, taking into account the scalability and power efficiency of the solution. To do so, it leverages the polyhedral model, a mathematical representation of the code that allows to perform static analysis and transformations on polyhedral codes. The information gathered by polyhedral analysis is then used in order to implement the basic block that the final architecture is going to use, called Stencil Streaming Timestep (SST). Finally, the design is automatically synthesized and the bitstream of the circuit, as well as a firmware to test the hardware, are generated.

1.4 Structure of the Thesis

The thesis is structured as follows. Chapter 2 introduces the state of the art of Electronic Design Automation (EDA) tools to give an overview of all the abstraction levels used by EDA designers. I begin with describing the Register Transfer-
CHAPTER 1. INTRODUCTION

Level (RTL) and conclude introducing the Electronic System-Level (ESL) design, the research area this work belongs to. The mainstream ESL tools are then evaluated and compared with the work done in this thesis. Chapter 3 recaps the context, the state of the art, and explicitly states the problem this thesis targets. It also briefly explains the given contribution. In Chapter 4 there is a brief overview of the core polyhedral analysis concepts required for the comprehension of the rest of the work. The architectural model of the accelerator used by the framework – Streaming Streaming Timestep (SST) – is presented in Chapter 5. The workflow of the framework presented in Chapter 5 is then automated by a toolchain thoroughly explained in Chapter 6. The chapter introduces the issues faced in order to automate the architecture derivation process and describes the solution found. Chapter 7 presents three test cases, showing the results achieved through the framework. Lastly, Chapter 8 summarizes the thesis’ work and discusses potential future improvement of the tools.
Chapter 2

State Of the Art

This chapter gives an overview of the main tools used by FPGA designers during the architecture design process. The Electronic Design Automation (EDA) tools are going to be introduced, giving an idea about what they are and how they fit inside the design flow. The entire process of designing an FPGA architecture is going to be analysed, highlighting what are the main level of abstractions involved in the process. Is going to be shown as well, how the design flow changed over the years introducing higher level of abstraction that enable a faster prototyping of the architectures.

2.1 FPGA Overview

Field Programmable Gate Array (FPGA) is a technology that was born in the mid 1980s. The main idea behind the FPGAs is to have a reprogrammable device that could replace different of Application-Specific Integrated Circuits (ASIC). With this in mind Freeman, Vonderschmitt and Barnett created in 1984 a startup, Xilinx, that started the production of the XC2064, the first FPGA, in May 1985 [17]. Since then the FPGA market started to expand and it has seen the birth of other companies producing FPGA. For sure another company to mention is Altera that, with Xilinx, is holding the majority of the market. The chart below
shows how the FPGA market was divided mostly between the two companies.

A FPGA consists of an array of Configurable Logic Blocks (CLBs) linked through programmable interconnects. This structure enables them to be programmed to a desired application or functionality. Every CLB is made of a configurable switch matrix, selection circuitry and flip-flops. It is extremely adaptable: it could implement combinatorial logic, shift registers or RAM. The input and output signals of the CLB are routed through flexible interconnect that also link the gate array with the external interfaces [18]. The FPGA has two possible modes, configuration mode and user mode. When it is powered up the circuit is in configuration mode: it is inactive and has to be configured to implement a design. The configuration consists in downloading a stream of bits, called a bitstream. It encodes the information about the function that every CLB has to implement and how they need to be routed. Once the configuration is done the FPGA is in user mode, it is active and is implementing the application specified in the bitstream configuration [19].
2.2 Pre-HLS design flow

The usual design flow for electronic circuits in the past two decades was obliging the hardware designer to define behaviorally the system at the Register Transfer Level (RTL). The RTL is a abstraction used in hardware description languages (HDL), such as Verilog and VHDL, that allows the representation of a digital circuit. An RTL description of a synchronous circuit usually contains two kinds of elements: the registers, that are elements having memory properties and are use to synchronize the circuit operation, and the combinatorial logic, which is digital logic created using Boolean circuits and is time-independent. Below there are an example of a RTL representation of a digital circuit and the respective representation in VHDL language.

![RTL representation of a digital circuit](image)

**Figure 2.2: RTL representation of a digital circuit**

```vhdl
D <= not Q;
process(clk)
begin
  if rising_edge(clk) then
    Q <= D;
  end if;
end process;
```

**Listing 2.1: VHDL example**
Clearly, having to specify all the details of a digital circuit at the level of the Boolean logic, is a really long and error prone task. For those reasons a higher levels of abstraction have been introduced in the design flow. Once the RTL definition is complete the designers take advantage of many Electronic Design Automation (EDA) tools. The function of those tools is to automatically perform a translation of the circuit specification, using lower level representations, and to find solutions to optimizations some optimization problem that arise when choosing a particular implementation of a design. It is possible to see a parallelism between how a compiler handle an input source file in order to generate the respective executable file, and how the EDA tools handle high level specification and translate them in lower level specifications. After that the functionality of the circuit is described using the RTL notation, the design flow is completed running the RTL synthesis and the place and route. The RTL synthesis lowers the level of abstraction to the gate level, in which basically the circuit behaviour is specified using a netlist. The netlist contains all the information regarding the elements that will be part of the final circuit, that are called instances, as well as the way that those instances are interconnected. The placement step takes all the instances contained in a netlist and assigns exact locations on the board to each one of them. This is a critical step because a bad placement not only can impact considerably the performance of the circuit, but it could make the problem of finding a suitable routing, which is the next step in a digital design, unsolvable. The routing step, taking into account the placement of the instances, does all the required connection between the elements. The place and route is a delicate phase, because the solution found to the problem of placing the instances on the board, and to the problem of routing those instance, has to take into account the timing of the circuit. The timing analysis is a type of analysis that ensures that the combinatorial logic propagation times are in line with the clock period, therefore guaranteeing that the output of the logic is corrected. All of those steps
aren’t performed by hand, and are part of Electronic Design Automation (EDA) tools. In the case of FPGA, a programming file has to be generated. The programming file encodes all the information regarding the allocation of the resources on the board, and regarding how to wire those resources together. Using this file the board can be configured in order to implement the desired circuit. The entire process of obtaining the programming file, required to configure an FPGA in order to implement a certain function, from an RTL behavioural specification goes under the name of logic synthesis. There are several commercial tools, that fall in the category of electronic design automation tools, that aim at the automation of the logic synthesis.

2.2.1 High Definition Synthesis tools

The term high definition synthesis is the term used to group all the operation that are involved into transforming a HDL representation of a design in a low level circuit representation. The new representation, after being analyzed by place and route algorithms, produces a bitstream configuration that can be used to configure an FPGA. This section is going to give an introduction of the main commercial software used for logic synthesis. Some of those software are complete synthesis solution that are FPGA vendor specific, for example Vivado and Quartus II, that can respectively be used for Xilinx’s and Altera’s board. Other are multi-vendors softwares, able to target boards produced by different companies, but focus on a particular point of the synthesis, like Synplify.

Vivado by Xilinx

Vivado Design Suite is a framework created by Xilinx for synthesizing HDL designs on the Xilinx’s FPGA boards, and to run analysis and simulation en-
abling optimization over an HDL specification. It was released in 2013 replacing the 15 years old Xilinx ISE, that until than has been the solution adopted in order to configure the Xilinx’s FPGAs. Vivado synthesize a VHLD, Verilog, SystemVerilog or mixed language design to creates a gate level representation. It is possible to setup and start a synthesis process in two main way, a Project mode and a Non-Project mode. Used in Project mode Vivado handles all the project files and provides a Graphical User Interface (GUI) that allows the designer to manage the circuit elements and connection through a block design. The Non-Project mode, allows the user to use the command line and to create scripts in order to setup and synthesize a design. The language used by the tool in this mode is the Tool Command Language (Tcl). The two mode are strictly connected, every action performed in the Project mode GUI has a respective tcl command that can be used in tcl scripts and using the command line [20].

Quartus II integrated Synthesis by Altera

Altera Quartus II is a design software produced by Altera. It can be used to create FPGA and structured ASIC design from HDL designs. It supports VHDL and Verilog design and performs in a comprehensive environment synthesis, place and route, timing analysis, simulation and configuration. Altera also developed its proprietary digital HDL language called Altera Hardware Description Language (AHDL). It can be used as an alternative or in conjunction with VHDL and Verilog for specifying the design of a digital circuit. Quartus II has three different user interface that can be used for each phase of a design flow: the graphical user interface, the EDA tool interface and the command-line interface. The EDA tool interfaces allows the designer to use external Electronic Design Automation tools integrating them in the graphical user interface or with command line executables. The command-line interface enables the designer to run Tcl commands and to create and run Tcl scripts [21].
**Libero SoC by Microsemi**

Libero System on Chip is a software toolset for designing with Microsemi, previously known as Actel, FPGA and SoC FPGA. It combines Microsemi’s design tools with EDA tools. The management of the application projects as well as the programming software are done by Microsemi’s proprietary software. This involves the phase of project creation, HDL editing and FPGA configuration. The phase of synthesis and of simulation are handled by third party software. The synthesis of the HDL languages is left the external software, Synplify Pro developed by Synopsys. The simulation is handled by ModelSim developed by Mentor Graphics [22].

**Synplify by Synopsys**

Synplify Pro is an FPGA synthesis software developed by Synopsys producing high-performance FPGA designs. It takes VHDL and Verilog languages as input HDL specification and synthesize them into a respective gate level representation. The most interesting feature of this tool is its ability to target FPGA from different vendors. It supports Xilinx, Altera, Microsemi, Achronix and Lattice boards. In order to do so, after the synthesis phase its over, the bitstream is generated integrating vendor specific place and route software [23].

**LeonardoSpectrum by Mentor Graphics**

LeonardoSpectrum is a synthesis tools that targets FPGAs and ASICs. Like Synplify it targets FPGA produced by different vendors, integrating FPGA vendor place and route tools. It implements hierarchical support for incremental synthesis allowing the re-synthetization of individual RTL level modules while preserving the netlist information of the surrounding block. A peculiar feature of LeonardoSpectrum is that is platform independent permitting the designers to
perform error-free retargeting of designs from FPGA to ASIC [24].

2.3 High Level Synthesis

High Level Synthesis nowadays is a popular approach in Electronic Design Automation, that enables the hardware designer to use a level of representation more abstract than RTL when designing the circuits. A more abstract view on the designing process can as well make easier to handle project with increasing complexity. It also speeds up the designer’s productivity by automating the translation of a behavioral description done at algorithmic level, using programming languages like C, C++ or Matlab, into a description at RTL level. Using HLS tools reduces drastically the amount of code that has to be written by designers, reducing as well the chances of making errors in the designs. High Level Synthesis is responsible of a serious of tasks that were usually performed by a designer. Resource allocation is the task in charge of determining, after an analysis of the operators and the memory elements described at the algorithmic level, what types of elements are needed and how many. This phase of the analysis does a trade off, between area of the circuit and performances, deciding what is the best configuration of the resources in order to minimize the area of the final solution, without impacting the performance of the circuit. The Scheduling assigns a clock cycle to each operation present in the algorithm; it tries to use as much as possible the available resources in order to obtain a schedule that performs the operation required by the algorithmic description in the minimum number of clock cycle. At last, the Resource Binding assigns each operation and data element to a physical operator or memory block. A part from speeding up the design process of a circuit HLS has also other advantages. It permits to automatically explore the space of the solutions for a given algorithm, a process that is called design space exploration, by modifying source code, without changing its semantic, and the tool options. HLS reduces the time required for the verification of the generated
CHAPTER 2. STATE OF THE ART

design because it can perform test on it reusing data on which it was tested the original source code [25].

2.3.1 High Level Synthesis tools

An overview of the main High Level Synthesis tool is provided in this section. Those tools vary on many aspect. One of the most important thing on which they differ is probably the source language that they analyze. The choice of the language is a crucial point in the realization of an HLS tool: it has to be flexible enough in order to ease the behavioral description, but at the same time it could have to impose some restrictions to allow the code to be implementable as hardware. Irregular control flow and complicated data dependencies can make a program not synthesizable. An other important characteristic of the tools is the design exploration capability. One input algorithm can in fact be implemented on hardware in many different ways. The tools differ highly in how they drive the exploration process, outputting solutions that may differ in area, performance or power usage. The most important directive that can be used to drive the RTL generation are loop pipelining and loop unrolling. The loop pipelining overlaps sequential loop iterations in the pipeline. Loop unrolling instead parallelize the body of some loop iterations. Lastly, the main point on the adoption of the HLS tool is to speed up the design production, so those tools need to have a fast learning curve.

Xilinx Vivado HLS

Vivado HLS is part of the Vivado Design Suite and is developed by Xilinx. Before being acquired by Xilinx it was previously known as AutoPilot and developed by AutoESL. The tool can handle behavioral specification done in C, C++ and System C. It outputs an RTL hardware description in Verilog, VHDL, optimized for area, latency and power consumption. As mentioned before, there are few constraints on those programming language that needs to be enforced
to enable the hardware generation. All the system call, as well as the Standard library calls are ignored by the compiler. The dynamic memory allocation needs to be replaced with static memory allocation. Not all the recursive functions can be implemented. The tool supports arbitrary precision types for all its input languages, allowing this way to model and synthesize exact bit widths. At the end of the synthesis process Vivado HLS shows a synthesis report. It contains information about the area, power and performance that the solution is expected to have once brought in Xilinx’s hardware. Once obtained the RTL representation of the algorithm, the solution given by Vivado HLS can be loaded in Vivado to be used in future designs [15] [26]. In 2015, probably in response to Altera SDK for OpenCL, Xilinx has released a new HLS tool SDAccel. This tool conforms with the openCL standards and it ease the implementation of accelerators that communicate over PCIexpress. However it targets only two device families, the Virtex-7 and the Kintex-7 [27]. Xilinx has also developed an other HLS tool called AccelDSP that now has been discontinued and is not available anymore. The interesting feature of this tool, that maybe is also the reason why is not available anymore, is that it was based on the MATLAB language [28].

BlueSpec Compiler

BlueSpec is a high level synthesis toolset developed by BlueSpec. It takes as input language a behavioral description of the architecture done in BlueSpec System Verilog (BSV). The compiler’s synthesis produces a Verilog RTL representation or SystemC executable. The Verilog representation can be used as input to 3rd party RTL synthesis tool to be used to configure an FPGA. The input language of the compiler is a synthesizable subset of SystemVerilog [29] which is an extension of Verilog. BSV should be able to provide a high-level of abstraction leveraging atomic transactions which it uses for managing complex concurrency. To be able to use this tool the algorithm has therefore to be completely reimple-
The compilation process can be controlled by the BlueSpec GUI or from a command line interface. One of the interesting aspects of this compiler is that the source code is scanned line by line to generate RTL code, preserving the original signal name. Those two factors improve the readability of the code generated. The tool offers no design exploration capabilities mainly because the BSV description is already at hardware level [25] [30].

Altera SDK for OPENCL

Altera SDK proposes a solution for high-level synthesis that uses as input definition the OpenCL C language. It is a C-based language that has OpenCL extensions that facilitate extraction of parallelism from the code. The target system of the SDK is an heterogeneous architecture composed by an host processor that communicates through PCIe with kernel's accelerators. OpenCL has also some Application Program Interface (API) using which the host can communicate with the accelerators over PCI Express and kernels can use to communicate between each other without the host interaction. A peculiarity of the approach of Altera is that the FPGA image is directly produced in one step, instead of outputting an RTL description that has to be further synthesized. The compilation of the FPGA kernel needs to be done separately from the host source code. The host source code can be compiled using GCC or Microsoft Visual Studio. The kernel source code has to be compiled using the Altera Offline Compiler. The FPGA board is configured directly at runtime using OpenCL dedicated functions [31].

LegUp

LegUp is an high-level synthesis software created by the University of Toronto and is open source. What is interesting about this tool is that it is implemented as a back-end pass to the LLVM compiler infrastructure. This implies that all the source code analysis and optimization are done by the LLVM C compiler and
LegUp takes as input the optimized LLVM Intermediate Representation and generates an RTL behavioural description from it. The output language that they use for the RTL description is the Verilog language, their output can be targeted to any FPGA vendors thanks to the use of generic dividers, RAM blocks and multiplier. The majority of the LegUp code is implemented within a top-level class called LegupPass that gets called by the LLVM pass manager [32].

**Catapult C**

Catapult is a HLS synthesis tool developed by Mentor Graphics in 2004 and acquired by Calypto Design System in 2011. The tool takes SystemC and ANSI C++ behavioral descriptions. After analyzing the behavioural description it synthesize a multi-block pipelined and concurrent hierarchical RTL design that can be later targeted to ASIC or FPGA. The RTL specification can be produced in VHDL, Verilog or System C, in conjunction with simulation and synthesis script for ModelSim and other simulation tools. Catapult C enables the designer to do design space exploration producing X-Y plots and bar chart with detailed comparison between different solutions. It generates from the C++ source code a hierarchical Grantt chart that provides information about data flow, and loop execution profiles. The tool uses symbolic analysis and optimization techniques like sequential constant propagation, array index analysis, bit width analysis and memory bandwidth optimization [33].

**Clash**

Clash is a functional hardware description languages that uses the semantics of Haskell, a functional programming language. It is the product of a research started at the University of Twente. The advantages of using a functional language to describe hardware come from the fact that the combinatorial logic of a circuit can be directly modeled ad mathematical functions. Those languages also
have properties like purity and single binding behavior that simplify program verification the application of code transformations and optimizations [34]. The CLaSH compiler that is installed as a cabal package, can compile the language, producing VHDL code, Verilog code and SystemVerilog.

C-to-Silicon Compiler

C-to-Silicon is the HLS solution developed by Cadence. It reads C, C++ and System C input source files and, after the analysis, it generates a Verilog RTL micro-architecture. It supports Xilinx and Altera FPGA devices. Among the feature of this compiler there is the Behavior-structure-timing (BST) databases allowing incremental synthesis and direct mapping of physical design data back to RTL as well as the original source code. It also provides a cross-linked graphical design environment showing control-dataflow graph, critical paths, area and power tree maps [35].

ImpulseC Impulse C compiler is a HLS tool developed by Impulse Accelerated Technologies. The handled input language is called Impulse C. It is a subset of the standard C language with a library supporting parallel programming. The architectural parallelism is obtained by implementing the solution using different processes that support data flow and message-based communications. The compiler fully supports Xilinx’s and Altera’s boards [36].

2.4 Electronic System-Level design

To increase the designer’s productivity the trend of abstraction from the hardware is getting to the last abstraction level of the hardware design: Electronic System-Level (ESL). The majority of the tools that claim to be ELS tools, only perform C-to-RTL high level synthesis. True ELS synthesis needs instead to implement a complete flow that is able to generate directly a complete system from
AN algorithmic specification. Those kind of tools usually focus on handling a restricted kind of domain specific application, relaying on some sort of computational model. The architecture that they generate is on the other side a heterogeneous or homogeneous multiprocessor system on chip (MPSoC) solution. The ELS follows generally a top-down approach that can be put in correspondence with software implementation steps. A way to visualize this link between hardware and software implementation is by the double roof model [37]. This model shows the software design process on one side and the hardware design process on the other side. It is divided in levels of abstractions and every vertical arrow represents a synthesis step, where a specification is transformed into an implementation. Every horizontal arrow on the other hand represents change in the level of abstraction.

![Double Roof Model](image)

**Figure 2.3: Electronic system design flow**

The ESL synthesis is in charge of selecting a suitable platform architecture and bringing a behavioural description of an application on that platform. The synthesis process starts therefore with a behavioral description of the application to implement and a set of constraint that define implicitly or explicitly a platform
model. The synthesis transforms the specification in an implementation. This is made of a structural model and quality numbers [38]. The first is a mapping of the behavioral model into the given platform, and is as well an architecture composed of different basic-blocks. Each of those block is going to serve as input for the next synthesis step.

2.4.1 Electronic System-Level tools

The Electronic System-Level is still a young field. As so, it’s boundaries are not yet well defined and it changes frequently and rapidly. In this section some of the tool that are commonly considered ELS tools are going to be briefly examined.

Daedalus

Daedalus is an automated framework that takes a sequential application as behavioural specification and produces an MPSoC system implementation on a FPGA. The framework considers only applications that can be modeled using the Kahn Process Network (KPN) Model of Computation [39]. In order for a C source application to be translated in a KPN model it need to be a static affine nested loop program (SANLPs) [40]. The architecture that Daedalus considers is an MPSoC where software processors and hardware cores exchange data using distributed memory units. The memory unit are implemented by FIFO that guarantee the synchronization of the processing unit using blocking read and write primitives. Daedalus design flow is divided in three phases. Each of those is handled by specific submodules: KPNgen, Sesame and ESPAM. KPNgen takes a sequential c program and represents it in a concurrent KPN specification. This specification can be translated into equivalent KPNs automatically enabling this
way a certain level of design space exploration. Sesame taking the generated KPN is in charge of performing the design space exploration. It outputs a high level platform description and process binding description that, together with the original behavioral KPN descriptions, are used as input to ESPAM. ESPAM produces the VHDL code that can be synthesized on FPGA using the vendor’s synthesis tools [41].

![Figure 2.4: Daedalus design flow](image)

**System-On-Chip Environment (SCE)**

System-On-Chip Environment (SCE) is a system-level design framework that implements a top-down refinement-based design targeting heterogeneous systems. SCE is based on the SpecC language and methodology [42]. It uses a *specify-explore-refine* methodology [43]. The first phase, the (*specify*) begins with a behavioral model specifying the design functionality. In the *explore* phase, the tool explores the design spaces taking necessary design decisions. Then the *refine* phase creates a model by adding the design decision to the previous model. At
the end of each refinement the result is a Transaction-Level Model (TLM) [44], a behavioral description that separate the details regarding the communication from the details regarding the computations. After the final design is fully specified the synthesis phase implements the hardware and software components. For the software processors some binary images are generated, and for the hardware core their Register Transfer Level specification allowing to synthesize and manufacture the MPSoC [45].

![System-on-chip design flow](image)

**Figure 2.5: System-on-chip design flow**

**SystemCoDesigner**

SystemCoDesigner is an ELS framework that aims at automatically mapping SystemC application to heterogeneous MPSoC platforms. The desired behaviour of the architecture is written in an actor-oriented application model by the designer. From the SystemC model the framework automatically generates hardware accelerators for the actors and stores them in a component library. The platform model is defined by the designer specifying the resources from the component library as well as mapping constraints for the actors. After the auto-
matic design space exploration produces a set of optimized solution from which the designer can choose. The architectural solution is represented as a structural Transition-Level Model. For rapid prototyping the accelerators can then be specified in RTL and the software compiled following the ISA of the computing units in the MPSoC. The framework assumes that the application model is written using SysteMoC [46], a library actor-oriented for SystemC, where the actors, that implements the functionality using a single thread, are communicating between each other only using SysteMoC FIFO. The process of the actor hardware synthesis is composed of three steps. The SysteMoC actors are first translated into SystemC modules. The modules are then synthesized in a Verilog representation, using the HLS tool Forte Design System Cynthesizer. In the last step the Verilog netlist is further synthesized in a netlist using Synplify Pro from Synplicity [47].

MAMPSx

MAMPSx is a design framework that generates a Heterogeneous Multiprocessor System-on-Chip (HMPSoC). It extends a older work called MAMPS in which each computing element could only be a homogeneous general purpose processors. The framework uses Synchronous Data Flow Graphs (SDFGs) to model a concurrent media application. As demonstrative output, it generates a prototype of the architecture that targets the Xilinx Zynq ZEDboard [48]. An SDFG \((A,E)\) is made of a set \(A\) of actors and a set \(E\) of edges. Every edge \(e = (a_1, a_2, t_1, t_2)\) represents a dependency between actor \(a_1\) and \(a_2\). The edge \(e\) contains two tokens \(t_1\) and \(t_2\); when an actor fires it executes its task. An actor can fires if there are enough tokens on all its input edges and enough space in the buffer of its output channels. When an actor fires, it uses tokens from its input edges and produces tokens on the output edges. An edge can contain the initial tokens, indicated by a bullet point.

MAMPSx uses C-HEAP to describe the communication interfaces, trying to
achieve decoupling of communication from computation. It uses a circular buffer with synchronization primitives. The data producer *claim space* on empty buffer space, and *release space* to release the claimed space. On the other hand the consumer has to *claim data* and *release data* to get read a data buffer and to release it respectively. The other input that the framework takes is the architectural template. It describes the processing elements available in the architectures, the *tiles* and their connections, the *interconnect*.

In order to map an application described by the SDF graph to a given plat-
form the tool uses SDF3 [49]. This tool, in charge of analysis and solution explo-
rations, defines buffer distributions, task mapping and static-order schedules. In
the last step MAMPSx generates a HMPSoC from the output given by SDF3. The
tiles (e.g. ARM and accelerators ) and the interconnect ( e.g. FIFO), specified in
the SDF3 mapping are instantiated, then the relative software projects are gen-
erated for each processing tile. The result of the above operation is then ported
automatically to target the Xilinx Zynq ZEFboard [50].

\section{PolyFPGA}

The work presented in this thesis is a framework that belongs to the category
of the Electronic System-Level tools. It is written to specifically target FPGAs and
to carefully leverage their resources. Moreover, the tool provides scripting that
allow the designer to completely automate the phase of FPGA configuration.
Using .tcl files to drive the Vivado Design Toolchain, the framework is able to
automatically perform all the steps in order to bring the application behavioral
description to the board. This is opposed to Daedalus as this toolchain forces
the designer to have an interaction with the HLS tools in order to synthesize
its VHDL output. Other that that, the Deadalus framework targets generic MP-
SoCs; for this, it uses FPGAs as simple prototyping platforms, and the resulting
FPGA implementation is not optimized. The framework presented in this work
allow the description of the input application using normal C code. SystemCoDe-
signer forces the use of SystemC to specify the behaviour of the application. This
forces the designer to rewrite the algorithmic description using an actor-oriented
model. Likewise MAMPSx needs to have an algorithmical description using the
Synchronous Data Flow Graphs.

This thesis’ work, in contrast with other ESL tools, allows to easily and quickly
specify the application behavior using statically defined C. Additionally, it explicitly produces a dataflow architecture optimized for FPGA that can explicitly scale to multiple nodes, allowing a certain level of design space exploration, and is able to customize the design around the platform at hand.
Chapter 3

Problem Statement and Contribution

This chapter is going to define the problem that this thesis addressed in the first section. The second section describes the contribution given by the proposed work.

3.1 Problem Statement

High Performance Computing (HPC) is a really active research area. As it has been discussed previously, other than being itself a field of research it also has effects in countless other areas of study. Being able to perform computations at high speed enables the use of always more detailed mathematical model, that make it possible to produce more accurate results and predictions. The evolution of HPC brought petascale computers in 2008; researchers are now trying to realize exascale computers.

However, few issues must be solved before we actually realize such systems. First and foremost, we need to improve the power efficiency of future large scale computing systems. Previous researches show evidence that the communication infrastructure of an HPC system is one of the place where most of the power is
consumed. For this reason, augmenting the data locality and the data reuse in a computation can significantly reduce the amount of communications in a computation and therefore its power consumption. On the other hand it has been shown as well that a dedicated architecture is more power efficient if compared to a general purpose processing element. For what has been explained above it seems reasonable to think that an architecture comprising reconfigurable hardware that can be reconfigured ad-hoc for each target application could perform better, under the power efficiency metric, than the current systems.

The second problem relates to the complexity of hardware design. In the previous chapter it was given an overview of the configuration process for an FPGA. It was shown that, because of the complexity of the process, the designer tends to rely on Electronic Automation Tools (EDA). Those tools, shifting the abstraction level from the hardware to the behavioral level, allow to design quickly larger systems. Few of them however focus on the Electronic System Level (ESL) of abstraction providing a fully automated way to design an FPGA. Tools at this level of abstraction are the key to create large systems based on FPGA.

Finally, within the HPC area the scientific computation are known to be often regular and repetitive. The majority of the computing time is spent inside some loop cycle that executes one instruction or a block of instructions, accessing data in a regular way. The fixed structure of those kind of programs is easier to analyze and optimize if compared to any other normal application. The fact, for example, that the data are accessed in a regular way can be leveraged preparing the data before they are needed by an instruction. Those codes are also suited to be modeled using a powerful mathematical representation, the Polyhedral Model. This representation allows to perform many loops transformations, exact data dependency at instruction level and memory access analysis.
3.2 Contribution

Within this thesis’ work I approach the problem of improving the power efficiency of future large scale computing systems by presenting a fully automated framework, PolyFPGA, that allows to perform hardware design at the Electronic System Level for a class of scientific algorithm called Iterative Stencil Loops (ISL), with particular attention to the way the resulting architecture is composed. PolyFPGA takes as input a C source code and an architectural template and produces a bitstream that can be used to program either a Zybo or a V707 board. The tool allows to describe the application behavior using C, one of the most widespread programming languages. Using polyhedral analysis it retrieves exact information regarding data dependencies and memory accesses. With those information the tool tailors a Streaming Stencil Timestep (SST) architecture that accelerates the code representation given as input. The SSTs are then combined in a complete FPGA architecture, that is automatically synthesized using the Vivado Design toolchain. PolyFPGA also predicts the maximum SST queue length thanks to an estimation algorithm.
Chapter 4

Polyhedral Model

In this chapter I introduce the core concept of polyhedral modeling. Section 4.1 briefly describes the mathematical background behind polyhedral analysis and the terminology used in the area. Section 4.2 is about code transformations in general. Section 4.3 is about transformations through the polyhedral model. In this section legal transformations and data dependences are explored. Section 4.4 shortly discusses the unimodular model, which is strictly contained in the polyhedral model and allows to represent and apply code transformations using matrices. In the last section the classes of code that can be analyzed using the polyhedral model are overviewed.

4.1 Introduction

In the process of compiling a program, before obtaining an executable file the source is usually translated in one or many intermediate representation languages (IR). Examples of IR could be gimple [51], used by the gcc compiler, or LLVM-IR used by the LLVM compiler [52]. The IR is useful to perform program’s optimizations that would be impossible to realize by direct manipulation of the source code, or using a lower level representation, and simplifies the process of translating a code from source language to machine language (instructions/op-
codes). There are many transformations that can be applied to the code, aiming at restructuring it, or at changing the data layout, improving the way the information are accessed and used. The problem of choosing the transformations to apply to a program in order to optimize certain metrics (like: code size or code speed or data locality, just to name a few) is non trivial. The Polyhedral Model is a mathematical framework that allows to model code loops and manipulate them in a sound manner in order to improve some of the resulting code’s metrics. It uses the mathematical concept of Z-Polyhedron [53] to represent each statement enclosed in an affine loop nest. Transformations are then performed directly on the polyhedron, using functions that map each integral point inside the original polyhedron into a point in the transformed polyhedron. After performing all the required transformations, the resulting polyhedron can be reconverted into the corresponding loop.

4.2 Background

The first part of this section contains a summary of the basic mathematical concepts needed to describe the polyhedral model. The second part, instead, describes the basic definitions and terminology. In the last part I will give a brief overview of the polyhedral tools we will use throughout this work.

4.2.1 Mathematical Background

I will first introduce the necessary mathematical concepts and toolings in an informal way to gently approach the matter; afterwards, for a more rigorous presentation of those concepts, the reader is suggested to review [53] and [54].

Affine Function

A function \( f : \mathbb{K}^m \rightarrow \mathbb{K}^n \) is affine if there are a vector \( \vec{b} \in \mathbb{K}^n \) and a matrix \( A \in \mathbb{K}^{n \times m} \) such that:

\[
\forall \vec{x} \in \mathbb{K}^m, f(\vec{x}) = A\vec{x} + \vec{b}
\]
CHAPTER 4. POLYHEDRAL MODEL

**Affine Space**

Given a set of vector, the set that is closed under affine combinations of those vectors, is called *affine space*. Given an affine space defined starting from a set of vectors, its *affine sub-space* are all the affine spaces contained in it such that their elements can be defined using a lower number of dimensions.

**Affine Hyperplane**

In an m-dimensional *affine space* an *affine hyperplane* is an m-1 affine sub-space. An affine hyperplane divides an affine space in two affine *half-spaces*, each of them can be represented using an affine inequality.

**Polyhedron**

A set $P \in \mathbb{K}^m$ is a polyhedron if there exists a system of a finite number of inequalities, each of them representing an affine half-space, $Ax \leq b$ such that:

$$P = \{x \in \mathbb{K}^m | Ax \leq b\}$$

Therefore it is the intersection of a finite number of half-spaces.

**Parametric Polyhedron**

Given n the vector of symbolic parameters, P is a parametric polyhedron if it is defined by:

$$P = \{x \in \mathbb{K}^m | Ax \leq Bn + b\}$$

**Polytope**

A polytope is a bounded polyhedron.

**Lattice**

A *lattice* is a subset L of $\mathbb{Q}^n$ generated by the integral combination of a finite set vectors $a_i \in \mathbb{Q}$. $L = L(a_1, \ldots, a_n) = \lambda_1 a_1 + \ldots \lambda_n a_n | \lambda_i \in \mathbb{Z}$ If $a_i \in \mathbb{Z}$ then L is an *integer lattice*.

**Z-Polyhedron**

A *Z-Polyhedron* is the intersection of a polyhedron with an integer lattice.
4.2.2 Terminology

In the following section an informal definition of terms that will be used in the rest of the work will be given.

**Statement**

A *statement* is a computer directive that will be interpreted as an action to perform.

**Operation**

A statement represents the description of the action to perform, this action could be executed many times if the statement is placed inside an iterative program structure, each instance of a statement is called an *operation*.

**Instruction**

The operation itself will be interpreted by the compiler and translated in multiple machine-level *instructions*.

**Loop Nest**

Each statement can be repeated iteratively when it’s inside an iterative program structure like a loop. A finite set of nested iterative loop is called *loop nest*. The *depth* of a loop nest is defined as the number of nested loops in a loop nest. A *loop iterator* is a variable linked to a loop that is incremented at the end of each iteration of the loop it belongs to, by a non-zero variable called *step*. A loop iterator takes as starting value the *lower loop bound* and as ending value the *upper loop bound*. An *iteration vector* for a loop nest is a vector containing all the loop iterations of the nest in order.

```plaintext
for(i=0;i<100;i++){
  for(j=0;j<70;j++){
    statement;
  }
}
```

Listing 4.1: Loop Nest

In the example above there is a loop nest containing two loops, it has therefore a depth of 2. The loop iterator of the outer loop is the integer variable *i* that ranges
from 0, the lower loop bound, to 100, the upper loop bound. The loop iterator for the inner loop is the integer variable \( j \). Both the loops have a step size equal to one. The iteration vector for this loop nest is the vector \((i,j)\) that contains the loops iterators of all the contained loops.

**Iteration Domain**

The iteration vector is a column vector containing in order all the iterators that are part of a given loop nest. The size of this vector is the same of the loop nest’s depth. Considering now the iteration vector as an \( n \)-dimensional variable, where \( n \) is the depth of the loop nest, the domain of this variable is going to be \( n \)-dimensional.

The iteration domain is going to be the set containing all the possible values of the iteration vectors, taking into account the loop constraints. Referring the loop nest 6.2, the iteration vector of that loop nest is the 2-dimensional integer vector \( \vec{i} = (i,j) \). This vector lies on a 2-dimensional space, that is the iteration space of this loop nest. The iteration domain of that loop nest is a set of 2-dimensional point defined by the loop bounds:

\[
\begin{cases}
  i \geq 0 \\
  i \leq 99 \\
  j \geq 0 \\
  j \leq 69
\end{cases}
\]

That can be represented in matricial form as:

\[
D = \{ \vec{x} \in \mathbb{Z}^2 | A \vec{x} + \vec{a} \geq \vec{0} \}
\]

Where:

\[
A = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 10 & -1 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} i \\ j \end{bmatrix}, \quad \vec{a} = \begin{bmatrix} 0 \\ 99 \\ 0 \\ 69 \end{bmatrix}
\]

**Lexicographic Order**

The lexicographic order is used to order the different operations that are inside the same iteration domain. If all the loop steps of an input program are are
positive variables then we can define an order between the integral points of a
domain: Taking the operation with coordinates \((a_1...a_n)\) from an n-dimensional
space, this operation is executed before an other operation with coordinates \((b_1...b_n)\)
iff exists an integer \(i\) such that the entry \(i^{th}\) of the iteration vector of \(a\) is less than
the corresponding entry in the iteration vector of \(b\) and all the other entry \(k<i\) of
both iteration vectors are equals.

\[(a_1...a_n) << (b_1...b_n)\]

Given vector \(\vec{0}\) that is the vector with all component equal to zero, a generic
vector \(\vec{v}\) is said to be lexicographically positive if \(\vec{0} << \vec{v}\).

**Static References**

Given a reference to an array cell \(B[f(\vec{x})]\), \(f\) is going to be called the subscript
function. If the subscript function is affine it can be expressed as \(f(\vec{x}) = F\vec{x} + \vec{f}\)
where \(F\) is the subscript matrix and \(\vec{f}\) is a constant vector.

\[B[i + j + c_1, j + c_2] = \begin{bmatrix} 1 & 10 & 1 \end{bmatrix} \begin{bmatrix} ij \end{bmatrix} + \begin{bmatrix} c_1c_2 \end{bmatrix}\]

A reference is said to be static if its subscript function is affine and it depends
only on outer loop counters and formal parameters.

### 4.3 Program Transformations

Two programs are said to be semantically equivalent when they guarantee
the same outputs regardless of their input data. This concept is useful when its
required to estimate if a transformation, once its applied to a program changes
its semantic. A given program may present some constraints among its statement
that force particular order on the execution of some of its statement. Those con-
straints are the program dependencies. The most commonly accepted dependen-
cies conditions are the berstein conditions [55]. Following Bernstein definitions,
three kind of groups can be identified: write after write (or output) dependen-
cies, write after read (or antidependance) dependencies and read after write (
or true dependencies). We say that two statements have a WAW dependency if both make a writing access to the same data, that they have a WAR dependency if one performs a write operation on a data that is later read by the other, and that there is a RAW dependence if a statement reads from a data that is later written by an other statement. It is sufficient for a transformed program to maintain the same dependencies between the statement that the source program had for them to be semantically equivalent.

There are many kind of transformations known in the litterature useful in order to optimize a program with respect to a target criteria. Some of those focus in optimizing the data, their layout, the memory required to store them. Others focus into applying changes to the source code itself. Given a particular program only a subset of those transformation can be applied, the subset that satisfies the dependencies constraint.

4.3.1 Reordering Transformations

A reordering transformation don’t add or remove statement from the source code, it modifies the just the order of execution. Because the statements obtained after a reordering transformation are the same, no new dependency is created and all the original dependencies are preserved in the transformed code. However, the direction and distance of a given dependency might change ad a result of a reordering transformation, even if the source and sink of all dependencies are maintained.

Fundamental Theorem of Dependence [56]

A reordering transformation which maintains all the dependencies of a program preserves its behaviour.

Given the theorem above a transformation is going to be called legal if all the dependencies in a program are preserved.
CHAPTER 4. POLYHEDRAL MODEL

4.4 Dependencies Representation

Many ways are present in the literature to represent a program’s dependencies, from more simpler ones as distance vectors, direction vectors, dependence vectors [57] [58] to more complex and exact ones such as dependence polyhedron [59] [60].

Restating the definition of dependence: A statement $S$ is in a dependence relation with a statement $U$ if there is a variable $x$ of $S$, a variable $y$ of $U$, an iteration $(i, i_s)$ in the iteration domain of $S$, and an iteration $(j, j_r)$ in the iteration domain of $U$, such that 1. one or both of $x$ and $y$ are output variables of their statement; 2. in their respective iterations they both point at an identical memory location; 3. in the serial execution of the program, the instruction $S(i, i_s)$ is computed before $U(j, j_r)$ of $U$; and 4. during the same computation, $M$ is not accessed for writing while $S$ finishes and $U$ starts.

Dependency of this kind impose constraints on the execution order of a program.

4.4.1 Distance Vectors

Each dependence can be represented as a distance vector; given a dependence between iteration $\vec{p}_1$ and iteration $\vec{p}_2$ of generic statement, the distance vector that represents this dependence is defined as $\vec{d} = \vec{p}_2 - \vec{p}_1$. Considering the fact that the lexicographic ordering of the program have to guarantee correct execution given that it represents the order of the initial program, it can be deduced that if there exist a dependence between two iterations $p_1$ and $p_2$ then their lexicographic order must be $p_1 << p_2$. So, vector $\vec{d} = \vec{p}_2 - \vec{p}_1$ have to be lexicographically positive. The ordering constraints between different iterations of a statement can therefore be expressed by a set of direction vector defined as follows: $D = d|d = p_1 - p_2$ and there is a dependency between $p_1$ and $p_2$. This implies that taken two statements, if $s_1 = s_2 + d$ where $d$ is a direction vector belonging
to the set of direction vector containing the dependencies of the program, then \( s_2 \) depends on \( s_1 \). It should be noted that this representation might implicitly introduce extra arcs because taking an iteration \( \overrightarrow{p} \), and a distance vector \( \overrightarrow{d} \) belonging to the set of dependency then for all \( \overrightarrow{c} \) there exist a dependency between \( \overrightarrow{p} \) and \( \overrightarrow{p} + \overrightarrow{c} \).

4.4.2 Direction Vectors

Not all the dependecies can be represented as a finite set of distance vectors, that is why the concept of direction vectors have been introduced. A direction vector captures only the information regarding the direction of a dependency, so each dimension of a direction vector has the following informations: + ( or > ) if the direction of the dipendence is positive. - ( or < ) if the direction of the dipendence is negative. 0 otherwise. Therefore a possibly infinite set of distance vectors can be represented by a direction vector. As an example, lets take the infinite set of distance vectors defined as follows :

\[
D = \{ \overrightarrow{d} | \overrightarrow{d} = (x, 0, 0) \text{where} x \in \mathbb{N} \land \text{x!} = 0, \}
\]

so the set of all distance vector with a positive value on the first dimension and a zero on the other two. This set can be represented as one direction vector, ( + , 0 , 0). It’s important to note that, even with this representation and for the same reason explained above, all direction vector have to be lexicografically positive for a program to be legal.

4.4.3 Dependence Vectors

The dependence vectors are a generalization of the direction vectors and distance vectors[58]. A dependence vector in an n-dimensional nested loop is a vector \( \overrightarrow{d} = (d_1, ..., d_n) \). Its components \( d_i \) are a possible infinite range of integer \([d_{\text{min}}, d_{\text{max}}]\) where \( d_{\text{min}} \in \mathbb{Z} \cup \{-\infty\} \) and \( d_{\text{max}} \in \mathbb{Z} \cup \{\infty\} \). Each of those vectors therefore represent a set of distance vectors. This set contains all the distance vectors that have components contained in the range of the dependence vector. In a
degenerate case when $d_{\text{min}} = d_{\text{max}}$ the dependence vector is a distance vector. Adding to the range of the dependence vectors the infinite value allows the representations of direction vectors. The ranges $[1, \infty]$, $[-\infty, -1]$, $[-\infty, \infty]$ correspond in fact to the direction vector $+$, $-$ and $0$ respectively.

### 4.4.4 Dependence Polyhedron

The dependence polyhedron representation is the only exact dependence representation presented in this section. It is a polyhedron that is in a space built on the cartesian product of the dimension of the spaces were the two statements belong. Each integral point in this polyhedron represent an existing dependence between the two evaluated statements and has coordinates $(s_1, \ldots, s_n, r_1, \ldots, r_m)$ where $s$ and $r$ are the two statements respectively belonging to a loop nest of depth $n$ and $m$. The $s$ coordinates are coordinate of the point in the iteration domain of statement $s$ that is in dependence relation with the point represented by the $r$ coordinates in the iteration domain of statement $r$. To check if there is a dependence between two statement using a dependence polyhedron, first the dependence polyhedron is computed\[59\], then an emptyness check is done on the resulting polyhedron. If the polyhedron is empty, then there are no dependencies between the two statements, otherwise each integral point inside the dependence polyhedron is a dependency between the statements. The emptyness check can be done using PIPlib, first defining the polyhedron, then computing the lexicographic minimum of the polyhedron\[61\]. If the polyhedron doesn’t have a lexicographic minimum, then its empty.

### 4.5 Unimodular Model

The Unimodular Model allows to apply the following loop transformations, and any combination of those: loop permutation, loop reversal and loop skewing. To use this framework in order to apply those transformations the source
code needs to be represented in its iteration space (see 4.1). The iteration domain of each statement that is contained in a loop nest is therefore represented by a system of linear inequalities, identifying in this way a set of integral points that represent all the iteration of that statement. After that the code is represented by its iteration space, any one of the transformations targeted by this framework can be represented by a unimodular matrix, so by a matrix whose determinant is either +1 or -1. Applying the transformation identified by the unimodular matrix to the integral points of a statement, we are performing a linear mapping that associate each integral point from the starting iteration space to one on the transformed iteration space. The reason why the matrix is unimodular is that each unimodular matrix has an inverse, and its inverse is unimodular as well, and that the transformation preserve the volume of the initial iteration space in the transformed space.

4.6 Class of programs

The use of the polyhedral model is restricted to certain classes of program that enjoy of some properties. During the years researchers put effort into extending those classes of program in order to let more problems tractable by the polyhedral model.

The first class of programs to be addressed was introduced by a work of Feauturier and later on called Static Affine Nested-Loop Programs (SANLP).

4.6.1 Static Affine Nested Loops (SANLP)

A Static Affine Nested-Loop Program (SANLP) [62] is a program where each statement is enclosed by one or more loops and if-statements, and where:

- loops have a constant step size.

- loops have bounds are affine expressions of the surrounding loop iterators,
static parameters and constants;

- if-statements have affine conditions in terms of the loop iterators, static program parameters, and constants.

- array references are affine functions of iterators and parameters.

- the passage of data between statements of the loop is visible to the compiler.

The interest in this class of programs lies in the fact that their behaviour is completely defined at compile-time, allowing, among other things, exact dependence analysis. Generally scientific computations have a regular structure therefore they can easily be specified respecting the SANLP requirements. The first extension to the SANLP where the Static Control Parts Programs (SCoP) is used. The main difference with the previous class of programs is that SCoP loosen the requirements of the code allowing for dynamic parts in the code, that however won’t be taken into account by the polyhedral model. From the starting code therefore is selected the set of SCoP of that program, each SCoP is then translated into the polyhedral model alone, and is as well optimized by it without considering the rest of the program.

4.6.2 Summarization of the properties of Static Control Parts Program (SCoP)

The following is the list of properties that programs must respect in order for them to belong to the SCoP class [59]:

- loops have affine bounds and if have affine conditions;

- the only data structure are matrices accessed using affine subscript functions;

- loops and conditions only depend on outer loop iterators and parameters;

- subroutine and function calls have been inlined.
It’s possible to see that the two definitions above are closely related, even if the SANLP class of program is strictly included in the Static Control Parts Programs.

### 4.6.3 Extensions to the SCoP class

The successive extension of the SCoP is the class of the **Weakly Dynamic Programs**, that in short extends the SANLP allowing the use of dynamic data inside affine if conditions, introducing this way a weak dynamic. This approach, even if it extends the number of programs tractable with the polyhedral model, has as downside the fact that the exact dependence analysis, done with the dependence polyhedron, have to be substituted with a conservative, inexact analysis Fuzzy Array Dataflow Analysis [63].

The last extension of the class of programs handled by the polyhedral model is the class of **Affine Nested-Loop Programs with Dynamic Loop Bounds**, that is translated first to a weakly dynamic program, before the polyhedral representation [64].

### 4.6.4 Properties of an ISL

The class of algorithm called ISL is strictly contained in the SANLP algorithms [65].

The ISL codes are iterative algorithms that given a matrix and an update function, called **transition function**, update every cell of the matrix. This matrix update process can be performed one or more times. That’s why they are called iterative, and the number of updates is specified by the outermost loop that defines the so called **time dimension**. A single step, within the time dimension is called **time step**.

One of the most important things to note is that the transition function, to compute the matrix update values, performs access to the original matrix with a fixed pattern that takes the name of **stencil pattern**.
The code above is an example of 5-point Iterative Stencil Loop. In this code the time-dimension is represented by the loop having $t$ as iterator. The timestep is then the piece of code contained in the time loop, that performs a single full matrix update. This ISL has a bidimensional iteration space. Its iteration domain is given by the $i$ and $j$ loops, and is defined by the following set of inequalities:

$$1 \leq i \leq 8$$

$$1 \leq j \leq 8$$

In figure 4.1 the data domain of the input matrix, which is 10 by 10, relative to the code in listing 4.2 is shown in grey. The area showed in cyan is instead the iteration domain obtained by the two inequalities above.
The iteration domain of an ISL can be mapped using a matrix access, resulting in a new data domain. In the figure below is shown a 5-point stencil pattern. All the five points shown in the picture needs to be read in order to compute every new data of the output matrix.

![5-point, 2D stencil pattern](image)

**Figure 4.2: 5-point, 2D stencil pattern**

### 4.7 Polyhedral Tools

A brief overview of some of the mainstream polyhedral tools is going to be given in this section. The presented tools are all open source and are used by PolyFPGA, the framework presented in this work.

**Clan**

Clan (Chunky Loop ANalyzer) [66] is a free software and library written by Cedric Bastoul. It translates pieces of high level codes in C, C++, C# or Java in the OpenScop representation [67], which specifies them using the polyhedral model. This specification can be used as input for other tools that are able perform code analysis and optimization. This tool is for example integrated into PoCC and Pluto high-level compilers.

**Candl**

Candl (Chunky Analyzer for Dependencies in Loops) is a free software and library written by Cederic Bastoul. It analyzes the OpenScop polyhedral representation. It can performs exact data analysis on a given polyhedral representation as
well as analyzing the legality of a polyhedral transformation. It gives as output a
dependence graph, showing all the dependencies found between the statements
present in the given model, or a violation graph, if it is asked to analyze the le-
gality of a polyhedral transformations.

**ISL and ISCC calculator**

ISL (Integer Set Library) [68] is a thread-safe C library that manipulates sets and
relations of integer points bounded by affine constraints developed by the INRIA
university. The ISCC calculator is a software that uses the ISL library, allows to
perform polyhedral operations such as computing the lexicographical minimum
of a set, computing the map of an integer set for a given mapping function, com-
puting the intersection, union and difference of polyhedral sets. The ISL library
performs computations in exact integer arithmetic using GMP.
Chapter 5

The Streaming Stencil Timestep

This chapter is about the Streaming Stencil Timestep, an accelerator template that allows to compute Iterative Stencil Loop algorithms in hardware. Section 5.1 presents the accelerator and its modules. Section 5.2 discusses the relevant components of the Streaming Stencil Timestep. Section 5.3 describes the high level methodology used to construct a custom accelerator for a given input ISL code.

5.1 Introduction

The hardware acceleration, is a technique that allows to offload part of the computation of a computing system to a specifically designed hardware. Given a SCoP source, that is therefore fitting in the polyhedral model, it is possible to know exactly the computational flow during its execution. This gives the ability, as explained in chapter 4, to perform exact data dependency.

The information regarding the flow of data in the code can be used in order to generate a streaming architecture, tailored over an ISL code, that will perform in hardware, therefore achieving much higher throughput, the same operations that are done in the software. Extending the methodology found in [69] is possible to generate such architecture automatically.

To generate the streaming architecture the time-dimension of the ISL, at the
beginning, is not taken into account. The core module of the architecture is the SST that performs one entire execution of the loops contained in the time dimension. Once the structure of the SST is defined, the final memory architecture is obtained by queuing many SSTs together.

![Complete architecture obtained queuing SST modules](image)

**Figure 5.1: Complete architecture obtained queuing SST modules**

### 5.2 Streaming Stencil Timestep

The SST is an hardware component that is in charge of executing a single matrix timestep update. The SST component performs one computation of the transition function at each iteration. (see 4.6.4). There are four main component in the architecture of an SST: filters, kernels and demux, and fifo memories.

The stencil access pattern is basically hardcoded in the SST architecture and defined by the disposition of the filters of the SST. All the filters that read data from the same matrix are grouped together in channels, and ordered in inverse lexicographical order (4.2.2). During the SST computation, each different matrix accessed by it is first linearized, then streamed in the respective channel. There is a filter for each read access to the matrix required in order to perform one computation of the transition function, and between each couple of filters there is a FIFO memory. The size of this FIFO memory depends on the size of the original input matrix and on the stencil pattern.
5.2.1 Filter

A filter is a component of the SST that has, in the general case, three ports: one input port that will receive the ILS matrix, and two output ports, a filtering output port and a fifo output port.

Considering the data domain of the entire matrix, the subscript function of a matrix access can be seen as a mapping from the original data domain of the ISL matrix to an image data domain. The function of a filter is to stream on its filtering port the cells that belong to the image data domain.

The whole isl matrix is first linearized, then streamed to a chain of filters. Each filter is connected to the next one making the FIFO output port of the first feed in the input port of the next one, passing through a FIFO memory. Those filters are obtained from the original code analyzing the reading accesses done on the original matrix. At each SST computation they feed the data required to perform the computation to the kernel module.

The FIFO memories that link two filters need to be of the same data size as the lexicographic distance between the two filters linked. This way, the matrix’s data between the two respective access that will be present due to the matrix
linearization, will be stored in the FIFO, and at each computational step both filters will be able to stream on their filtering output port the data needed for the computation.

5.2.2 Kernel

The kernel module is the component in the SST in charge performing the computation. It reads data from the filtering ports of the filters, and outputs the newly computed data on its output port.

5.2.3 Demux

The demux module reconstruct the matrix taking into account the ghost zones (or halo) created by the stencil computation. In order for the SST to stream from its output a matrix of the same size of the input one, the demux fills the ghost zones using the values contained in the original input matrix.

The polyhedral model, after analyzing a stencil code, allows to compute what cells will be in the ghost zones. The demux will be connected to the central filter, in charge of streaming the entire original matrix, and to the output port of the kernels. Then it will, accordingly to the ghost zones analysis, stream data from the original matrix, or from the kernel, to have an output matrix of the same dimension of the input one.

5.3 The Methodology

This section is going to describe at a high level of detail the methodology that allows to obtain automatically a Streaming Stencil Timestep for a given ISL code. The main Intermediate Representation (IR)s used through the tool are also going to be introduced here.

The methodology is composed by three main parts: extraction, intermediate representations and synthesis.
5.3.1 Extraction

During the extraction phase the snipped of code to accelerate in hardware are extracted from an input regular C source. The whole source code is parsed and the part of code between some pragmas are extracted to be further analyzed. The central role in this phase is done by the clan library, that has implements the parser and gives as output a polyhedral representation of the code.

After the extraction phase the code passes through a three Intermediate Representations (IR). Each intermediate representation step bring the code information closer to the final SST architecture.

5.3.2 Intermediate Representation

The first IR is the exact data dependency graph of the SCoP we are analyzing. It is computed using the candl library, that analyzes the output obtained by the clan library, and outputs a directed graph containing all the data dependencies occurring between the statements. The graph is then manipulated, all the dependence arcs that don’t represent a RAW dependence and all the dependence arcs that represent a dependence over the temporal dimension of the ISL are removed.
This is done mainly because the only dependencies of interest in a dataflow architecture are the RAW dependencies, and because what is going to be modeled is the flow of data for a single timestep, so the dependencies along time dimension are not useful.

After the removal of the unneeded arcs, a graph where each node represents a statement present in the ISL and each arc represents a RAW dependency between two statement within a timestep is obtained. This graph is then transformed into a streaming oriented graph.

**Streaming Oriented Graph (SOG)**

A streaming oriented graph is a graph in which each node represent an access to a matrix done by a statement. The access can be either a reading either writing access, and each arc, that connects two accesses, represents how the data is moving from one access to the other. A read access node is therefore connected with an arc to the respective write access node. All the arcs computed by candl concerning the RAW dependencies are preserved and used to link the respective accesses that are the cause of the dependencies.

In order to transform the exact data dependency graph into the streaming oriented graph each statement node of the data dependency graph is substituted with the graph of its accesses. All the reading accesses performed by the statement become a node in the new graph, and all of them are connected to the only write access of the statement.

The obtained graph is likely to contain the so called *copy dependencies*.

**Copy Dependencies** A copy dependency is an arc that goes from a write node to another node. It represents a mere passage of data between a chain of nodes, that has only the effect of delaying the propagation of the information.

There are few constraints that a dependency has to satisfy for it to be considered a copy dependency:

1. The write node has to have only one outgoing arc.
2. The landing node of the arc does not have to have any other arc that connects it to the first writing node.

3. If the landing node is a read node its data domain have to be equal to the iteration domain of the write node, while if the landing node is a write node the iteration domains of the two nodes have to be the same.

The streaming oriented graph is obtained after applying a node expansion algorithm and a copy-node removal algorithm. It is the skeleton of what will become the Streaming Stencil Timestep.

After the generation of the streaming oriented graph, it is translated again into the last intermediate representation, the streaming oriented architecture graph.

**Streaming Oriented Architecture Graph (SOAG)**

The SOAG is the last intermediate representation of an SST, it stores all of the information required for building the actual hardware architecture. Each node present in a SOAG models a piece of hardware that will be part of the final architecture, and each arc between two nodes models what will be a real wiring between hardware modules. Those kind of graphs hold more detailed information than the SOG graphs, for example the FIFOs that are put between the modules of the final architecture, the channels from where the matrices will be streamed in the SST, and the demux are taken into account. During the construction of the SOAG graph there is the need to heavily rely on the information made available by the polyhedral tools. Those tools allow to decide the order the filters need to have within each channel to avoid deadlock in the final architecture, the depth of each FIFO that allows the filters to send a data to the kernel every cycle of clock. Moreover they are used to compute the equations that will tailor all the modules of the SST for the particular ISL taken into account. There is a series of steps to perform in order to obtain a SOAG from a SOG.

1. For each different matrix accessed in the ISL a channel needs to be instantiated. This channel will contain all the filters that take data from that same
matrix, and will be connected to the DMA that will stream the entire matrix.

2. Within each channel all the filters need to be ordered in inverse lexicographical order to avoid deadlock that will halt the architecture during the computation.

3. The FIFOs connecting filters within the same channel have to be dimensioned in a way that for each clock cycle the filters can send data to the kernel; to do so the size of the original ISL matrix need to be computed.

4. The filter corresponding to the central access to the ISL matrix needs to be located, because this is the filter that will stream the border of the original matrix to the demux, allowing in this way the construction of an output matrix of the same size of the original.

5.3.3 Synthesis

The last step, the synthesis, will produce the desired architecture. In this step the information included in the SOAG will be used to generate a c HLS code, using Vivado conventions, for each filter, kernel and demux. Those HLS codes will be inputted into Vivado HLS tool, in order to obtain the respective IP core.

Following the connections in the SOAG, the IP cores generated by Vivado HLS will be connected to each other, and to the specified FIFOs. The resulting design will be create and packaged as a new IP. This is the IP that implements the SST.

All the required SST’s IP will then be included in the final architecture and using Vivado the bitstream is going to be generated.
Chapter 6

PolyFPGA Workflow

In this chapter I describe the workflow of PolyFPGA. The framework is divided into different modules, and the structure of the section follows in order the steps that bring from the input code to its accelerator. The first section examines the type of input code handled. The second section is about the parser which generates a polyhedral representation of the code and produces its Data Dependency Graph (DDG). The third section overviews the evaluation module, in charge of ensuring the correctness of the input. The fourth and fifth sections overview the intermediate representations used by the framework. In particular the fifth section also describes the main problems faced in the generation of the second representation. The sixth section presents the ISCC Interface, which is placed between the framework and the ISCC calculator, and allow the resolution of polyhedral queries sent by the tool. The remaining sections discuss respectively about the generation of the hls codes of the accelerator’s submodules, the creation of the ip core of the accelerator, the synthesis of the complete architecture and the generation of a firmware to test it.
6.1 Introduction

PolyFPGA, using the polyhedral model in order to perform static analysis and using the Vivado toolchain, is able to automate the whole process of generating a the SST based streaming architecture.

The tool, adapts the methodology found in [69], adding some steps that are mainly located at the end of the automation process.

From a high level of abstraction, PolyFPGA does the following:

1. Analyzes a suitable C source code that contains the specification of the tasks that will be performed by the final hardware and expresses those task specification using the polyhedral model.

2. Using the polyhedral information and the information gathered through the exact data dependence analysis, creates a first graph, for each task, that represents how the data will flow within the final architecture.

3. Performs some polyhedral queries to the ISCC calculator [68], then using the provided information, generates a second graph, that specifies completely the architecture that will be synthesized.

4. Generates high level synthesis C source file specifying the the behaviour of the main components of the SST architecture and a tcl file encoding how Vivado HLS can build those modules automatically, then runs Vivado HLS using those files obtaining the hls IP components.

5. Generates a tcl file that guides Vivado trough the automatic creation and packaging of the SST IP cores , based on the specification contained in the SOAG graph (see chapter 5). Using this tcl file runs Vivado, obtaining this way the desired IP cores.

6. Allows the user to perform a basic design space exploration using SST-queuing, a technique that replicates inside a chain an SST-accelerator in
order to apply the computation multiple times.

7. Generates a tcl file containing the instruction, requires by Vivado, to perform the automatic synthesis of the final architecture. The final architecture is created using a base architecture, that takes care of streaming data in and reading data from the IP of the SSTs, to which all the SST are linked. The tcl file guides Vivado through the process of synthesizing the bitstream.

8. In the last step, the tool creates a Vivado SDK project, using again custom tcl scripts, that includes the hardware previously synthesized. Then writes a c source code that aims at testing each SST IP included in the architecture, and builds the whole project. The result is an elf file that can be used as a testing firmware for the architecture.

Resuming the whole process, PolyFPGA takes as input argument a suitable C source file, and outputs, among other lesser important files, a bitstream to configure a FPGA, an elf binary file that can be used as a firmware for the architecture in order to test the functionalities of the SST IP, and a dot graph, for each SST implemented that represents in the form of a block diagram, how it was built.

The diagram shown below describes part of the workflow of the tool, underlining how the code has been divided into modules. Here it is possible to see the flow, starting from the extraction of the kernel’s code, ending with the generation of the last IR, the Streaming Oriented Architecture Graph (SOAG).

### 6.2 Input file

The input file accepted by PolyFPGA can be any C source file. To specify to PolyFPGA which part of the code is desired to be synthesized in hardware, it has to be surrounded by pragmas and be compliant with two requirements.

The code describing the functionality of the architecture, has to be first of all a static C code (see 4.6.1). This requirement has quite straightforward implications,
if the code have a behaviour that is defined at compile time, this means that the program flow does not change depending on the data given as input, and the series of instruction to execute is consistent, regardless of the inputs. This constraint gives the ability to obtain a simple and optimized hardware at the end of the PolyFPGA process, moreover it gives the possibility to leverage the polyhedral model, and the existing polyhedral tools in order to analyze the code.

The second requirement is that the implemented code have to be an ISL code. This requirement arises from the particular kind of end architecture that PolyFPGA synthesize, being it thought specifically for this class of code. Anyhow the tool was made in a way to allow future extension, so that if a new architecture model, able to extend the class of program addressed, is found, it can be included.

The listing below shows an example of code accepted by PolyFPGA.

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
```
#include <math.h>

static
void kernel_jacobi_1d_imper(int *A,
    int *B)
{
    int t, i, j;

    #pragma scop
    for (t = 0; t < 10; t++)
    {
        for (i = 1; i < 50 - 1; i++)
        for (i = 1; i < 50 - 1; i++)
            A[i] = B[i];
    }
    #pragma endscop

    static
void kernel_jacobi_2d_imper(int ***A,
    int ***B)
{
    int t, i, j;

    #pragma scop
    for (t = 0; t < 10; t++)
    {
        for (i = 1; i < 50 - 1; i++)
            for (j = 1; j < 50 - 1; j++)
        for (i = 1; i < 50 - 1; i++)
            for (j = 1; j < 50 - 1; j++)
                A[i][j] = B[i][j];
    }
    #pragma endscop

    static
void kernel_heat_equation(int ***A,
    int ***B)
{
    int t, i, j, k;

    #pragma scop
    for (t = 0; t < 10 ; t++)
    {
        for (k = 1; k < 50 - 1; k++)
            for (i = 1; i < 50 - 1; i++)
                for (j = 1; j < 50 - 1; j++)
        for (k = 1; k < 50 - 1; k++)
CHAPTER 6. POLYFPGA WORKFLOW

The example code shown is a regular C code, it can be compiled using any available C compiler. When it’s given to PolyFPGA as input, the tool is going to look for the snippets of code between the \#pragma scop and \#pragma endscop. In the example there are three kernel that are going to be analyzed, jacobi 1D, jacobi 2D and the heat-equation. The tool will proceed analyzing separately the three different kernels, until the SST IP of each of them is packed. After that, all of the kernel’s IP are liked to the same hardware project, from which then the bitstream is going to be generated.

The jacobi 1D and jacobi 2D kernels are taken in this example are a modified version of the kernel found in the polyhedral test bench polybench [70].

6.3 Parsing Module

The parsing module is the first module of PolyFPGA and it is in charge of initializing the state of the tool and of parsing the command line arguments and the input source code. The information that hold the state of PolyFPGA, is stored inside a data structure called option. This structure is instantiated by the parsing module at the begin of the program, and by all the function called by the tool.

First of all the parsing module stores the option that PolyFPGA received from the command line. Those options define the board type, and if the user interaction is enabled or disabled. Then the module instantiates the data structure that will hold the temporary information needed by the tool. All the procedures called

```c
for (j = 1; j < 50 - 1; j++)
for (i = 1; i < 50 - 1; i++)
    A[i][j][k] = B[i][j][k];
#endif
}
int main(int argc, char** argv)
{
    // Many uninteresting things.
}```
afterwords are going to store their results in the same data structure.

The data resulting from any analysis of a specific piece of ISL, are stored in a linked list in the same order on which the ISL codes appear in the source code. For each ISL contained in the given input code the parsing module extracts the respective polyhedral representation.

The clan library is a free library that represents parts of high level programs, C, C++, Java, C#, in polyhedral representation, called OpenScop[67]. This library is linked to PolyFPGA therefore, in order to extract a polyhedral representation from the given input code, the clan functions are used [66].

Below is shown an example code that is extracted from the original code and analyzed by clan.

```c
#pragma scop
for (t = 0; t < 10; t++)
{
  for (i = 1; i < 50 - 1; i++)
    for (j = 1; j < 50 - 1; j++)
  for (i = 1; i < 50 - 1; i++)
    for (j = 1; j < 50 - 1; j++)
      A[i][j] = B[i][j];
}
#pragma endscop
```

Listing 6.2: Jacobi 2D kernel

The parsing module uses the clan library function `clan_scop_extract` in order to extract the polyhedral representation of each ISL contained in the input code.

The invocation of `clan_scop_extract` returns a osl_scop variable. This data structure is defined in the OpenScop library, and holds static data analysis computed from the ISLs in the input code. Those information are than used as input for the candl library in order to compute the exact dependencies for each isl task in the given input code.

The graph shown below is the data dependency graph computed by candl with respect to the ISL given above.
6.4 Evaluation Module

The output of the parsing module is later checked by the evaluation module. This module is in charge of verifying if the given input code is compliant with the constraint given by PolyFPGA. Each test is self contained in one function that tests the input code for a specific constraint, for example to check the presence of parameters. Each one of those functions can then generate a normal error or a global error. When a normal error is generated the ISL task that is responsible for the generation of the error is rejected. The other ISL tasks are handled normally by PolyFPGA and the final architecture is going to be synthesized without the accelerator related to the task that caused the error. A global error on the other hand is generated when an error that prevents the generation of the final architecture is found. In this case PolyFPGA terminates without generating any architecture.

6.5 Streaming Oriented Graph Module

The Streaming Oriented Graph Module generates for each ISL code its first intermediate representation (IR). The module starts from the exact Data Dependency Graph (DDG), analyzes it, manipulates it, and generates the corresponded Streaming Oriented Graph (SOG). The IR represents the skeleton on which the ISL accelerator is going to be built. The SOG Module uses three algorithm to convert a DDG into a SOG: the edge removal algorithm, the node expansion algorithm and the copy dependencies removal algorithm.

First, the Data Dependency Graph is manipulated in order to retain only information about the flow of data within the ISL timestep computation. To do so
the unrequired dependency edges are removed from the graph.

The edge removal algorithm follows the condition described in [69] (Definition 4.3.1). An edge is removed if:

1. it represents a dependency carried along the time dimension, therefore going from node \( S_i \) to node \( S_j \) where \( j > i \)

2. it is a self dependency, therefore going from a node \( S_i \) to the same node \( S_i \).

The result of the algorithm applied to the graph of figure 6.2 is shown below.

![Figure 6.3: Data Dependency Graph after edge removal algorithm](image)

In this new graph the only dependence left holds the information regarding the flow of data within the statements of a given ISL.

The node expansion algorithm enriches the graph obtained in the previous step with the information regarding the statement accesses. Each node representing a statement is therefore substituted by a subgraph containing reading and writing information relative to the statement it is related to. For each reading access done by a statement, a reading node is created. A node is also created for the writing access done by the statement, and all the respective reading nodes are connected to it.

Below is shown the result of the node expansion algorithm applied on the graph in figure 6.3.

The reading node are represented in the graph above with the round nodes, the writing node are represented using squared nodes. The graph above is no
CHAPTER 6. POLYFPGA WORKFLOW

Figure 6.4: Streaming Oriented Graph with copy dependencies

...longer a DDG and it has lost the notion of statements and dependencies. It shows instead the data accesses and the relationship between them. The writing node will be, in the finalized accelerator, the computation units. Next to each writing node is shown the respective operation that it is going to perform in the accelerator.

At last, the copy node removal algorithm, removes some redundant datapaths from the SOG. A copy node is a node that merely pass data between other nodes, withoout altering them. It has only the effect of delaying the propagation of the information (see 5.3.2). The algorithm iteratively finds the copy nodes and removes them.

In the graph in figure 6.4 the reading node B[i][j] is a copy node. It is removed and the two connected nodes, the writing node B[i][j] and the writing node A[i][j], are merged. In order to merge the two nodes, their computation function is updated: all the reference to the no longer existing variable B[i][j] are substitute with the entire expression used to compute it.

The result obtained after applying this algorithm to the graph in figure 6.4 is the final SOG, and it is shown in figure 6.5.

The two writing node have been condensed in a single compound node, that combines the operations that they performed. This compound node will be translated at the end of the accelerator design into a processing elements. This process-
Figure 6.5: Streaming Oriented Graph

The Streaming Oriented Graphing element, reading all the data defined by the stencil pattern will generate at each computational step one output value of the output matrix.

6.6 SOAG Module

The Streaming Oriented Architecture Graph (SOAG) Module brings the intermediate representation closer to the hardware abstraction. In the SOG intermediate representation the object involved are reading nodes and writing nodes. Each reading node of the SOG is going to be translated into its respective filter and each writing node in the respective kernel. The goal of this step is to create a representation of the architecture detailed enough to be able to translate the single modules that it contains directly into a high level synthesis C representation. In order to do so its required to perform quantitative analysis on the Streaming Oriented Graph. Specially the following problems need to be addressed:

1. how many input channels is going to have the accelerator
2. how many input data is the accelerator going to receive
3. how much memory does the accelerator need
4. what is the order in which the accelerator is going to receive its input data

The result of these steps is going to fix the design of the Streaming Stencil Timestep accelerator.

The accelerator is going to have one input channel for each matrix it reads from. Even if the Iterative Stencil Loop has only one stencil input matrix on which it iterates, it can use in the computation data coming from other matrices. In the
final architectural design there a Direct Memory Access (DMA) is going to be instantiated in order to stream each matrix. Any input matrix is first serialized by the DMA, then streamed in the input channel and finally read by the accelerator. The SOAG module sorts the reading nodes putting all the nodes that reads from the same matrix inside the same input channel. The reading nodes can now be considered filters: they filter from the stream of data coming in the input channel the data required by the writing node, that becomes now the kernel of the computation. Then, in order to output a matrix having the same size as the input one, a Demux module is added in this representation.

There are two key information required to design each filter: the total number of data that they need to filter, and their filtering function.

### 6.6.1 Computing the Stencil Array Size

The number of data sent to filter is the same as the number of element contained in the input stencil matrix. Anyhow this information is not directly available because the input matrix is not defined in the code analyzed by the framework.

```c
#pragma scop
for (t = 0; t < 10; t++)
{
    for (i = 1; i < 9; i++)
    for (j = 1; j < 9; j++)
    for (i = 1; i < 9; i++)
    for (j = 1; j < 9; j++)
        A[i][j] = B[i][j];
#pragma endscop
```

Listing 6.3: Jacobi 2D 10x10

The Listing 6.3 shows an example of iterative stencil loop. The outer most loop, where t is the iteration variable, is the time dimension of the ISL; it defines how many update steps are performed on the entire input matrix. The firsts two innermost loops, having the i and j iteration variables, performs instead the
actual computation on the input matrix. The last two loops update the output matrix. The number of points in the iteration domain of the two innermost loops, having vector variable \((i,j)\), is not the same as the number of items in the input matrix; this is due to the stencil access pattern and the halo of the isl.

In figure 6.6 the iteration domain relative to the listing 6.3 is shown over the bi-dimensional matrix accessed. In order to compute the size of the input matrix the iteration domain is then mapped using the subscript functions of each data access done in the stencil loop to the input stencil matrix.

In figure 6.7 the iteration domain has been mapped using the subscript function of the matrix access \(A[i+1][j]\). The cell shown in yellow is the lexicographic maximum of the computed data domain. The lexicographic maximum of each of the resulting domains is computed, and the size of the matrix is inferred combining the the maximum element found for each dimension.
Figure 6.8: Computation of Array Size

Figure 6.8 shows how the size of the array is computed. Two different lexicographic maxima have been found holding respectively the maximum array size for each dimension. Their value is aggregated inferring this way that the input matrix is a square matrix 9 by 9.

6.6.2 Computing the FIFO sizes

In chapter 5 it was explained that, within the same input channels, the different filters are linked to each other passing through a First In First Out (FIFO) memory. This memory is going to store the data streamed in the input channel, waiting for them to be sent to the computing kernel by a filter. At each computation the data stored inside the FIFO moves of one position. Dimensioning this FIFO memory correctly is really important. If the FIFO is too small, then it won’t be able to store all the interval of data, defined by the stencil pattern, between the one being filtered. This will make the architecture stall, because the filters will be waiting for the lost data. On the other hand, if the FIFO is too big it will not arm the computation of the accelerator. However, the architecture is going to instantiate resources that it doesn’t need, making the final design bigger than required. Moreover, because the information will have to go through the whole chain of FIFOs, the architecture will have longer latency and consequently less
throughput than it could have had. In this section is going to be explained, in an informal way, how the formula to compute exactly the required FIFO size is derived. The final formula is going to be general enough to address matrices with any dimension. The problem is first going to be reduced to a more approachable one that is then going to be extended. Finding the exact size needed by a FIFO connecting two filters is in a way equivalent to finding the number of element between two cells of a matrix. This is justified by the fact that the distance between the data accessed by the FIFO its constant, even if at each iteration of the accelerators the filtered data change. In the case of a mono-dimensional matrix, the solution to the sub problem is straight forward. The number of elements contained between the cells $i_{th}$ and $j_{th}$ is given by: $\text{abs}(j_{th} - i_{th}) - 1$. In the case of multiple dimensions however the problem is going to be more complicate. To define an element inside a multi-dimensional matrix a positional vector having the same dimensions as the matrix is going to be required. So, for a bi-dimensional matrix, the problem becomes to compute the number of element contained in the matrix between two bi-dimensional vectors $i = (i_0, i_1)$ and $j = (j_0, j_1)$. The matrix is going to be streamed to the accelerator by the DMA module linearized. To compute the number of elements between the two vectors its therefore first required to linearize the two vectors index, after that the problem can be solved like in the linear case. The two linearized index will be then $i = i_1 \ast \text{DIM}_0 + i_0$ and $j = j_1 \ast \text{DIM}_0 + j_0$ where $\text{DIM}_0$ is the size of the dimension relative to the indices $i_0$ and $j_0$. The number of element between the two index is therefore given by $i - j = \text{abs}((i_1 \ast \text{DIM}_0 + i_0) - (j_1 \ast \text{DIM}_0 + j_0)) = \text{abs}(i_1 - j_1) \ast \text{DIM}_0 + \text{abs}(i_0 - j_0)$. In the most general n-dimensional case, the index linearization formula becomes:

$$
\sum_{k=0}^{n-1} (i_k) \ast \prod_{w=0}^{k-1} (\text{DIM}_w)
$$

where $i$ is the index vector defining a position in the N-dimensional matrix $\text{DIM}_{th}$ is the size of the n-th dimension of the matrix and $\text{DIM}_0 = 1$. 
CHAPTER 6. POLYFPGA WORKFLOW

Computing the distance between two elements linearized using the formula above the following it’s obtained:

\[
\sum_{k=1}^{1} ((j_k - i_k) * \prod_{w=0}^{k-1} (DIM_w))
\]

where \(i\) and \(j\) are the two vectors identifying the positions in a \(N\) dimensional matrix, \(j_{1n}\) represents the element at the \(n\)-th position of the vector and \(DIM_0 = 1\).

In order to use the previously derived formula to compute the size of the FIFOs connecting the filters the only thing missing is how to obtain a index vector, representative of the filter, from the filter’s subscript function. As it was said at the beginning of this section the distance between two elements filtered by the same two two different filters is constant over time, so any two filtered element can be used in the previous formula as long as they are filtered in the same iteration. For simplicity the first element sent to the kernel by each filter is going to be taken. That element can be found computing the lexicographic minimum of the data domain obtained by mapping the iteration domain with the filter’s subscript functions.

![Figure 6.9: Obtaining filter’s index vector: A[i][j + 1]](image)

Figure 6.9 shows the operation done to compute the filter’s identificative vector for the filter related to access \(A[i][j+1]\). The iteration domain is shown in cyan,
the data domain obtained by mapping the iteration domain using the access subscript function is the green area and in yellow there is the lexicographic minimum element: (1,2). Once obtained the two positions each one identifying a filter the previous formula can be used to compute the number of elements to store in the FIFO that links them and therefore the size of the FIFO.

### 6.6.3 Ordering the filters

Another important aspect that the SOAG module takes care of is the filter ordering. If the order of the filter is not following the inverse lexicographic order it is going to create a deadlock in the accelerator, the filter’s chain stalls and consequently the entire accelerator stalls. Therefore putting the filters inside their respective channels and in the right order is crucial to make the accelerator work correctly. As in the case of computing the stencil array size, this problem is solved recurring to the polyhedral analysis. To perform the ordering a positional vector has to be computed for each matrix access, then the respective filters are going to be ordered accordingly. To compute the positional vector for a particular filter the subscript function of the matrix access that it is related to is required. The iteration domain of the Iterative Stencil Loop is then mapped using this subscript function in a data domain. Lastly the positional vector of the filter is given by the lexicographic minimum of this data domain.

Referring still the example in Listing 6.3 in its first statement there are 5 access to the input stencil matrix $A$: $A[i][j]$, $A[i][j - 1]$, $A[i][1 + j]$, $A[1 + i][j]$, $A[i - 1][j]$. As it has been said previously for each one of those access a filter is going to be created in the accelerator, therefore there will be 5 filters to order. The steps described previously are performed using the subscript function of each of those access, to obtain the positional vector that is going to identify the filter order. Figure 6.10 shows the computation of the positional vector relative to access $A[i][j-1]$. The positional vector for this particular filter is then (0,1).
Figure 6.10: Filter ordering; access A[i][j - 1]

Figure 6.11: Filter ordering; access A[i][j - 1]

Figure 6.11 shows all the matrix accesses performed by the first statement in Listing 6.3 linked with their positional vectors. The access are ordered, from left to right in the inverse lexicographic order. The first filter in the input channel where the stencil matrix is streamed will then be the one related to to the access A[i+1][j]. On the other hand the last filter of the channel is going to be the one generated from access A[i-1][j].

6.7 ISCC Interface

The ISCC Interface is the part of the framework that performs all the polyhedral computation. The SOAG module heavily rely on it in order to get the information it needs to size correctly the architecture. This interface is put in between the PolyFPGA framework and the ISCC calculator, a tool that allows to perform polyhedral analysis leveraging the Integer Set Library (ISL). This calculator takes a problem representation done using the polylib specification language, on which it is able to perform most of the basic polyhedral operation. The type of operation
required by the PolyFPGA framework to work are mostly: set mapping, set difference, finding the lexicographic minimum in a set and set comparison. For each one of those operations there is a correspondent function in the ISCC Interface of PolyFPGA that is in charge of translating a more complex problem, described in terms of filters, kernels and demux, into many simpler set subproblems. Those subproblem are then defined using the polylib representation and then are sent to ISCC in order to to the computation. The result of the calculator is then translated and aggregated by the interface in order to provide a result to the framework.

6.8 HLS Generator Module

The High Level Synthesis (HLS) Generator Module creates HLS C source code for each component specified in the SOAG graph. To do so, the framework uses template C source files. There is an HLS template for each component of the architecture, meaning filters, kernels and demuxes. The parameters obtained by the SOAG Module are used to specify from the templates the accelerators specific HLS source file. The information regarding the dimensions and size of the input matrix as well as the filtering functions are used with the filter template to generate the source code of each filter. The compound statement generated in the SOG Module is used, with the information regarding its reading accesses, to create from the kernel template its source file. The information needed by the Demux in order to reconstruct the output matrix, done by adding the border’s data of the input matrix to the new data computed by the kernel, define with the demux template its source hls file. Once that all the source files have been generated, the HLS generator module produces the respective Vivado IP cores using Vivado HLS. The entire procedure its driven automatically by the framework. It is possible to instruct the framework to perform the high level synthesis directly on the machine where the framework is running, or to make it generate a script file that can be run on an other machine where the Vivado Design Tools
are installed.

6.9 SST Packager Module

In the previous module the vivado ip cores that will be part of the SST accelerator have been synthesized. A vivado project is created and then linked to the repository containing all those components. This module then uses the knowledge gathered from the SOAG in order to generate the full accelerator design. It drives the Vivado Synthesis tool automatically using a tcl script in order to instantiate the various element, that are after linked following the SOAG. After performing this steps, the vivado project is wrapped inside a single IP core that will therefore contain the entire accelerator. This IP, once wrapped can be included in other vivado designs.

6.10 Bitstream Generator Module

The Bitstream Generator Module is the part of the framework where the bitstream required to the configuration of the FPGA is created. The framework stores two architectural template each one specific to a Vivado evaluation board, a Zybo design and a Virtex 7 design. It can of course handle other boards if the required template designs are provided. The final architecture generated by this phase is mainly aiming at allowing to test the SST accelerator, as it will be explained in the future works chapter the architectural template can be improved. Each one of the two template design is leveraging an internal processin element that is in charge of providing the input stencil matrix to the accelerator through a DMA component and to handle the output coming from it. The zinx processor is an embedded arm processor that resides on the Zybo board and is the one used to perform the task just described. On the Vitrex 7 template design is instead used a microblaze as a processing element. It is a vivado IP core specifying a processor
that is then on the programmable part of the board. This module creates a new vivado project using the architectural templates described above, then for each SST input channel it instanciates a DMA component which is connected to the processing element. When the framework is executed from the command line it is possible to specifying a queuing parameter for each SST accelerator. Basing on this parameters all the different accelerators, coming from the different ISL specified in the original input code, are put inside an SST-chain. As an example, from an input code having three ISL codes, like the one shown in Listing 6.1, the framework is going to produce three different SST accelerators, one for each loop nest. For every single accelerator it’s possible to specify a queuing argument. The final design, is going to have three accelerators chain, each chain having a number of accelerators given by its queuing argument, connected to different DMA components. The processing element has therefore access to each one of those chains through the DMA components; the firmware binary of the processing element is going to define how and when to use those accelerators channel. This module automatize the design creation and synthesis phase using tcl script to give directive to the vivado synthesis tool. Once that all the required components are instantiated and connected to the template design, the synthesis is performed and the architectural bitstream is generated.

6.11 Firmware Generation

The last step performed by PolyFPGA is the firmware generation. The firmware is the software that will be used to program the processing element of the architecture, making it perform starting operation, such as allocating the memory for storing the input matrix and output matrix and initialize the input matrix and issuing the memory transfer to the DMA components in order to use the accelerators synthesized on the board. The firmware generated by the framework is a testing firmware aiming at showing that the accelerators as well as the complete
design are working properly. For each accelerators chain the firmware allocate and initialize a matrix of the right dimensions and size. Then those matrices are sent to the hardware accelerators and the result of the computation is printed out.

6.12 Conclusions

In this chapter the most interesting operation performed by the framework have been explained. The whole process started with a input c code with Iterative Stencil (ISL) Loops that needed to be accelerated. All the required steps in order to obtain SST based accelerators each given ISL code have been overviewed. The framework can create an accelerator or a chain of accelerators if the user required to perform SST-queuing. It was also explained how the framework is able to target different FPGA boards, the Zybo and Virtex-7, underlining the fact that, if provided with board templates, it can target many others. Lastly few words have been spent on the automatically generated firmware that allows the user to test the architecture.
Chapter 7

Experimental Results

In this chapter I will elaborate on the results obtained by running the framework on multiple stencil codes: jacobi 1D, jacobi 2D and the heat equation 3D.

7.1 Experimental Setup

The following test have been done using a specific experimental environment. Polytool was compiled and installed on a computer system having the following characteristics: OS: Ubuntu 14.04 64-bit RAM: 16 GB CPU: Intel® Core™ i7-2860QM CPU @ 2.50GHz × 4 C-Compiler: GCC 4.8.4

The testing machine needs to have a properly licensed installation of Vivado 2014.4 in the default path: "/opt/Xilinx". PolyFpga also relies on the following external tools: Clan version 0.8.0, installed in "/opt/polyhedral_tools/clan-install/" Candl version 0.6.2, installed in "/opt/polyhedral_tools/candl-install/" ISL version 0.14 installed in "/opt/polyhedral_tools/barvinok-install/" The PolyFPGA source files reside in the folder "/opt/clantool", and the framework was compiled in the same directory. The tests, as mentioned above, target the Zybo board and the Virtex-7 board. The Zybo needs to be powered on by one USB connected to the "PROG UART" port and the testing machine. To do so the jumper that specifies the power source needs to be set accordingly. In order to see the se-
rial output coming from the board the minicom program was used. The Virtex-7 was instead powered on externally, using the provided power cord. It was connected to the testing machine with two USB cables, one used to configure the board and one used to read the debug output. To see the board’s output the Vivado SDK console has been used.

7.2 Command Line Interface

PolyFPGA can run with different configuration given by the user. Below there is a table listing all the current options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-s 1</td>
<td>Synthesize architecture directly</td>
</tr>
<tr>
<td>-s 2</td>
<td>Generate synthesis script</td>
</tr>
<tr>
<td>-b 0</td>
<td>Target Zybo</td>
</tr>
<tr>
<td>-b 1</td>
<td>Target Virtex-7</td>
</tr>
<tr>
<td>-q &lt;num1&gt;,&lt;num2&gt;</td>
<td>Perform queuing of &lt;num1&gt; on first SST, &lt;num2&gt; on second</td>
</tr>
</tbody>
</table>

7.3 Test Input Codes

The tests have been run using three Iterative Stencil Loops. The first two benchmark are a monodimensional and a bidimensional jacobi algorithms. Those two kernels are part of the Polybench benchmark [70]. The third test case is the heat equation algorithm, a tridimensional kernel well known in the field of Stencil computations [71].

7.3.1 Jacobi 1D

The jacobi algorithm is a Stencil algorithm used to solve Laplace’s differential equations. This is the monodimensional algorithm, it has a 3-points Stencil access pattern. The input Stencil matrix contains 1080 elements.
CHAPTER 7. EXPERIMENTAL RESULTS

7.3.2 Jacobi 2D

This is the bidimensional version of the jacobi algorithm, it is a 5-points Stencil. The size of the matrix is 1920x1080, a full-hd matrix.

Listing 7.2: Jacobi 2D

7.3.3 Heat Equation

The heat equation algorithm is used to simulate the diffusion of heat in a tridimensional space. This Stencil code has a 7-points access pattern and the size of the input matrix is 100x100x100. The size of the matrix is limited by the use of the Vivado’s FIFO that have a fixed maximum size. The use of the FIFO memory grows exponentially with the dimension involved.

Listing 7.3: Stencil code for Heat Equation
CHAPTER 7. EXPERIMENTAL RESULTS

Listing 7.3: Heat Equation 3D

```c
#pragma scop
for (t = 0; t < 10 ; t++)
{
  for (i = 1; i < 100 - 1; i++)
    for (j = 1; j < 100 - 1; j++)
      for (k = 1; k < 100 - 1; k++)
}
#pragma endscop
```

7.4 Test Cases

This section is going to present the different tests done on the framework. The command line invocation of PolyFPGA employed to generate the bitstream is given for each test case, followed by a brief explanation of the settings given to PolyFPGA.

7.4.1 Jacobi 1D on Virtex-7

This test uses as input code the monodimensional jacobi algorithm in listing 7.1. PolyFPGA is set in order to target the Virtex-7 board using the apposite command line flag. For this test 5 bitstreams have been generated each one having a different queuing parameters.

The command used to launch PolyFPGA is the one below:

```
$ ./polyfpga -s 1 -b 1 -q 1 /opt/clantool/examples/jacobi_1D.c
```

Listing 7.4: Jacobi 1D PolyFPGA command

The -s is set to 1, so that polyfpga generates directly the bitstream instead of a synthesis script. The -b parameter is set to 1 to target the Virtex-7 board. The -q
is set to 1 to generate an architecture having one single SST accelerator. Then, to generate the other four bitstream PolyFPGA is run again setting \( q \) to 2, 8, 32 and 48; the rest of the argument are kept the same as the one showed in listing 7.4. The last algorithm is the path of the source file containing the Stencil loop in 7.4.

The graph in figure 7.1 shows the FLOPS obtained by the architectures on the y-axe over the number of SST used, in the x-axe. To compute the FLOPS the number of floating point operations done by the kernel to update the entire matrix has been divided by the time that it took the accelerator, or chain of accelerators, to perform the computation. The performance of the complete architecture increases linearly adding SST, on the other hand obviously the performance obtained by a processor is static when the matrix update is performed more times. As a reference the FLOPS obtained by a CPU Intel I7 at 2.53 GHz, running the same algorithm on a single thread, have been put on the graph.

![Figure 7.1: Performances Jacobi 1D](image_url)
Figure 7.2 shows in percentage with respect to the total availability, the use of the resources on the Virtex-7 board. The x-axe shows the SST used by the architecture while on the y-axe there are all the resources used by each solution as a percentage on the total available resources. The graph above shows that the resource usage increases linearly when adding SST accelerators. The more used one by the architecture obtained from the Jacobi 1D algorithm are the DSP followed by the LUT. With a chain of 48 SST 30% of the DSP available on the board have been used.

Figure 7.3: Power Efficiency Jacobi 1D
CHAPTER 7. EXPERIMENTAL RESULTS

Figure 7.3 represents the power efficiency obtained by the solution, measured as FLOPS/Watt. To estimate the power consumption of the architecture, the values given by the Vivado synthesis tool have been used. The x-axis on the graph is the queuing parameter given to PolyFPGA, while the y-axis represents the power efficiency obtained by each generated solution. The graph shows that the power efficiency increases linearly with the addition of SST accelerators.

7.4.2 Jacobi 2D on Virtex-7

The input code used in this test is the bidimensional Jacobi algorithm shown in Listing 7.2. The target board is still the Virtex-7 board, and like in the previous test, 5 architectures have been generated with different queuing values. PolyFPGA was launched using the same command shown in Listing 7.4, changing the input file path to point to the Jacobi 2D source file.

Figure 7.4 shows the FLOPS obtained by the architecture generated by the framework using different queuing parameters. These results are in agreement with the one obtained by hand, in a previous work [69]. This corroborates the results achieved by the framework and its validity. The graph above shows that the performance obtained by the SST architecture increases linearly with the addition of SST accelerators.
accelerators. On the x-axe there is number of SST used, and on the y-axes the performances obtained by each architecture.

Figure 7.5 shows the difference between the performance obtained by the SST architecture on the Jacobi 2D algorithm and the one of an Intel i7 @ 2.5GHz. On the x-axe there is the number of SST used in the design, for each SST architecture the performances given by a cpu running the same algorithm are placed next to the one obtained by the hardware accelerators. The performances obtained by the processors are constant augmenting the number of matrix update performed, while the performance of the SST architecture increase linearly adding SST IPs. While for low numbers of iteration the two are comparable the processors is outperformed from 8 SST.
Figure 7.6 shows the percentage of use of the Virtex-7 board for each generated architecture. On the x-axis the number of SST used is shown, while on the y-axes there is the use, in percentage, of the resources of the board. It grows linearly adding more SST to the design. In this case the most used resource were the LUT, with 48 SST on board over 40% of the LUT have been used.

Figure 7.7 shows the power efficiency obtained with the accelerator generated for the Jacobi 2D algorithm measured as FLOPS/W. The x-axis has the number of SST used by each solution generated by PolyFPGA, on the y-axis there is the
power efficiency obtained by each one of those architectures.

### 7.4.3 Heat Equation 3D on Virtex-7

This test uses as input code the one in listing 7.3. The Stencil has a 7-point access pattern. This kernel was synthesized for Virtex-7 with queuing parameters of 1, 2, 8 and 32. The synthesis of the architecture having queuing parameter of 48 failed because of the lack of BRAM on the board.

![Heat 3D Throughput Graph](image)

The graph 7.8 shows the performances with respect to the queuing parameters. The x-axes represents the number of SST used in the architecture, e.g. the given queuing parameter. For each queuing parameter the performance, in FLOPS, obtained by the SST architecture generated by PolyFPGA is placed next to the one obtained by regular processor. The results obtained on the same kernel by an intel i7 @ 2.53 GHz with a single thread is therefore showed next to the one given by the architecture.
Figure 7.9: Resources Heat 3D

Figure 7.9 shows the resources allocated for each architecture generated with different queuing values. On the x-axis there is the queuing parameter given to PolyFPGA to generate the architecture and on the y-axis there are the resources used by the solution obtained. It is possible to see that the architecture with 32 SST uses almost all the BRAM present on the Virtex-7 board, which is the reason why it wasn’t possible to generate an architecture with 48 SST.

Figure 7.10: Power Efficiency Heat 3D

The histogram in 7.10 shows the power efficiency given by the Heat 3D accelerator. The x-axes shows the different queuing parameters given to the PolyFPGA.
tool to generate all the different architectures. On the y-axes the is shown power-
efficiency, computed as FLOPS/W, obtained by each synthesized architecture.
Chapter 8

Conclusions and Future Works

In this work I presented PolyFPGA, a complete yet experimental tool to generate arbitrarily lengthy Streaming Stencils Timesteps queues. First I introduce the SST architectural template, and describe the benefits and limitations of this approach. Afterwards, I describe how to implement that methodology into a toolchain, which I then experiment with. Results are encouraging and demonstrate both the viability and the potential of this work.

The PolyFPGA framework is still an highly experimental framework, with much room for improvements. It needs to be further tested in order to ensure its robustness. Here some of the desired feature that can extend the capability of the framework are going to be overviewed.

**Input code extensions**

At the moment the framework is able to create hardware accelerators for a small class of input codes, the Iterative Stencil Loops (ILS) algorithms. This is mainly due to the choice of using the Stencil Streaming Timestep as a base to design the accelerators. In order to enlarge the class of code handled by the tool different base for accelerators could be added, so that each handled piece of code can be accelerated with its respective accelerator model. The polyhedral model is in fact able to handle a larger class of program, the static control programs, therefore
the intermediate representation currently used could be kept as is, adding just
the functionality required to handle different source algorithms. Other than this,
as it was discussed in section 4.6.3, research in the field of polyhedral analysis is
trying to enlarge the applicability of the polyhedral model, allowing the analyzed
code to have some dynamic and even nonlinear parts, an improvement that will
necessarily affect the way an architectural template is designed.

**Hardware Templates**

In section 6.10 the Streaming Stencil Timestep (SST) accelerators are connected to
the architectural template designs during the last steps of the framework. Those
designs use an on board processing element, used to host the computation, like a
zynq’s arm processor or a microblaze. The architectural templates currently used
for the Virtex-7 evaluation board could be substituted with on development of
a new architectural template that enables the employment of a different, more
powerful host processor that can stream the data to the board over PCI-express,
for the benefit of overall throughput improvement.

**Multi-FPGA**

A component capable of streaming data using Aurora links from one board of an
other has already been developed. This component can enable to split a SST-chain
on different Virtex-7 boards allowing the implementation of larger designs onto
multiple FPGAs. The framework could be then improved allowing to split the
design into sub-designs. For each sub-design the framework could then proceed
with the bitstream generation, obtaining a bitstream file for each used board.


Bibliography


[34] Matthijs Kooijman. Haskell as a higher order structural hardware description language. 2009.


