BOOSTING PERFORMANCE OF HLS OPTIMIZATION FOR SOC BASED HARDWARE ACCELERATORS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AZIZ BERKIN KOCAAY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JANUARY 2020

Approval of the thesis:

BOOSTING PERFORMANCE OF HLS OPTIMIZATION FOR SOC BASED HARDWARE ACCELERATORS

submitted by **AZIZ BERKIN KOCAAY** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Eng.** _____

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Supervisor, **Electrical and Electronics Eng., METU** _____

Examining Committee Members:

Prof. Dr. Gözde B. Akar
Electrical and Electronics Engineering,METU _____

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Eng., METU _____

Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering,METU _____

Prof. Dr. Ece G. Schmidt
Electrical and Electronics Engineering,METU _____

Prof. Dr. Ali Ziya Alkar
Electrical and Electronics Engineering, Hacettepe University _____

Date: 30.01.2020

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Aziz Berkin Kocaay

Signature:

ABSTRACT

BOOSTING PERFORMANCE OF HLS OPTIMIZATION FOR SOC BASED HARDWARE ACCELERATORS

Kocaay, Aziz Berkin
Master of Science, Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı

January 2020, 94 pages

Modern large-scale computing algorithms require huge amount of computational power. In adapting to increasing computation demands, FPGA-based SoC platforms provide an alternative to traditional CPU or GPU units, which suffer from thermal problems, power issues, etc. However, design flow for FPGA based development may be hard and time-consuming for an average software engineer who has limited knowledge about hardware design. A new approach in FPGA-based system development without the need for a hardware engineer is to program the FPGA using high level synthesis (HLS) tools that resembles C-based languages. Commercial HLS tools provide different kinds of automatic and user-defined optimizations for loop kernels such as pipelining, loop unrolling, etc. However, these techniques only provide instruction-level pipelining and reduce loop enter and exit overheads to decrease execution time of algorithms running on programmable logic (PL) side of SoC systems.

The limited approach of HLS for loop kernels can be extended by adding front-end operations to input code of HLS tools.

In this thesis, we propose a semi-autonomous polyhedral analysis and optimization-based methodology in order to enable course grained parallelization on nested loop

structures to increase final design efficiency. Xilinx Zynq SoC FPGA platform and Vivado Design Suite Tool are used in order to show how our proposed approach could be applied.

Keywords: Polyhedral Modelling, Hardware Accelerator, System on Chip (SoC), High Level Synthesis, Parallel Processing, Zynq

ÖZ

SOC TABANLI DONANIM HIZLANDIRMALARINDA HLS PERFORMANSINI YÜKSELTME

Kocaay, Aziz Berkin
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği
Tez Danışmanı: Doç. Dr. Cüneyt F. Bazlamaçcı

Ocak 2020, 94 sayfa

Yeni nesil, geniş kapsamlı hesaplama algorithmaları yüksek miktarda hesaplama gücüne ihtiyaç duyar. Bu artan ihtiyaca uyum sağlamak için, ısınma ve güç tüketimi gibi problemleri olan CPU veya GPU tabanlı platformlar kullanmanın yanısıra FPGA tabanlı SoC platformları kullanmak da bir seçenek olabilir. Fakat, geleneksel FPGA tasarım yöntemleri ortalama bir yazılım mühendisi için zaman alıcı ve zordur. Yüksek seviye programlama (HLS), FPGA tabanlı sistemleri geliştirirken, donanım mühendisi ihtiyacını ortadan kaldırararak, C-tabanlı dillerin kullanılmasına olanak sağlayan yeni bir yaklaşımdır. Ticari HLS araçları döngü çekirdekleri için otomatik olarak uygulanan veya kullanıcı kontrolünde olan bir dizi optimizasyon yöntemi sunar. Fakat, sunulan optimizasyon yöntemleri sadece komut seviyesinde ardışıklık sağlar ve döngü giriş ve çıkışlarındaki zaman kaybını azaltır.

HLS araçlarının bu sınırlı yaklaşımı, HLS aracının kullanımından önce giriş koduna bir takım işlemler uygulanarak genişletilebilir.

Bu tez çalışmasında, polyhedral model tabanlı analiz ve optimizasyon yöntemleri kullanılarak iç içe geçmiş döngülere paralellenebilme yetisi kazandırılmış ve HLS aracının giriş koduna uygulanmıştır. Bu yaklaşımın nasıl uygulandığını göstermek için

Xilinx firmasının ZYNQ SoC FPGA platformu ve Vivado Design Suite tasarım aracı kullanılmıştır.

Anahtar Kelimeler: Polyhedral modelleme, Donanım Tabanlı Hızlandırma, Çip Üzerindeki Sistem, Yüksek Seviye Sentezleme, Eş Zamanlı İşleme

To my family…

ACKNOWLEDGEMENTS

Foremost, I would like to express my deepest gratitude to my advisor, Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı for his excellent guidance and continuous support of my research.

I wish to thank ASELSAN A.Ş. for giving me the opportunity to continue my graduate-education and providing an appropriate environment to develop my thesis studies. I would also like to express my special appreciation to my colleagues and seniors from workplace for their contributions on the improvement of my engineering skills.

Last but not the least, I wish to thank my family who never hesitated from giving their supports to me.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS

AXI: Advanced Extensible Interface

CPU: Central Processing Unit

DFG: Data Flow Graph

FPGA: Field Programmable Gate Array

HDL: Hardware Description Language

HLS: High Level Synthesis

IC: Integrated Circuit

IP: Intellectual Property

ISL: Iterative Stencil Loop

I/O: Input/output

PL: Programmable Logic

PS: Processing System

SCoP: Static Control Part

SoC: System on Chip

CHAPTER 1

INTRODUCTION

In recent years, most of integrated circuit (IC) vendors around the world have released their powerful multicore heterogeneous CPUs, GPUs and CPU-FPGA hybrid systems [1]. Many modern applications which are based on machine learning, artificial intelligence, clouding demand huge amount of data processing in limited execution time [2]. However, efficient design for high-performance and low-power architectures for computationally intense applications is still not an easy task. Most system developers address this problem via transforming their algorithms to be executed on field programmable gate array (FPGA) platforms. FPGAs are charming solutions that allow implementing huge parts of target applications on hardware at low cost as they can serve better computational capacity, flexibility and lower-power consumption at the same time.

1.1. Motivation

According to Moore's law, transistor density in ICs has increased in the last decades [3]. As a consequence, many special functional blocks such as DSP units, on-chip-memories and CPUs have been added into modern FPGAs. Nowadays, they can be used as a combination of powerful programmable logic (PL) and flexible processing units (PU). Commercial and research projects, which employ this combined structure as an accelerator on different kind of applications, have shown remarkable improvement in terms of execution time when compared to traditional CPUs and GPUs [4].

However, the main disadvantage of the FPGAs is their more difficult programming and configuration procedure when compared to sequential instruction-based

programmable ICs such as GPU and CPU development. Traditionally, FPGAs are configured using a hardware description language (HDL), such as Verilog or VHDL whereas CPUs are programmed via one of a plethora of sequential programming languages such as C, C++ and Python. Although, theoretically both hardware description languages and instruction-based programming languages could be used to express any computation, there exists huge engineering workload differences between them. Traditionally educated software developers are usually not familiar with methodology of FPGA programming. Thus, software engineers have to learn low level HDL programming or cooperate with an expert hardware engineer, which increases the required human or time resources seriously in any FPGA based accelerator projects. The other way to use FPGA as an accelerator without requiring a hardware engineer is programming FPGA with high level synthesis (HLS) tools. HLS is an upcoming programming trend for FPGA programming with regular C-based languages [5]. HLS provides a higher-level abstraction of the system and removes details of traditional RTL based design to generate hardware architecture for a given target device.

1.2. Aim of the Thesis

HLS has been studied extensively in recent years and commercial HLS tools have reached a creditably stable level. Many researches in literature have shown that performance of hardware design generated by using HLS tools may reach to an acceptable level when compared with traditional RTL designs for non-complex algorithms [6].

Loop kernels are broadly used in computational-intensive algorithms. In SOC-based designs, FPGAs carry computational loads of systems because of its natural parallel processing capability. The efficiency of computation intellectual property (IP) mostly depends on the optimization techniques used during system design process. Commercial HLS tools provide different kinds of automatic and user-defined

optimizations for loop kernels such as pipelining and loop unrolling. These optimizations increase computational efficiency. However, these techniques cover only sequential instructions in loop bodies to enable instruction level pipelined or parallel architecture to enhance the performance of target algorithm.

This conservative approach of HLS can be broken by adding some front-end operations to input code of HLS tools. In this thesis, a semi-autonomous polyhedral analysis and optimization-based methodology is proposed in order to enable course-grained parallelism for computationally intense applications to reduce execution time due to natural advantages of parallel processing.

1.3. Contributions of the Thesis

The contributions of the present thesis work can be summarized as follows:

- A novel HLS design flow that integrates three different tools namely Xilinx Vivado Design Suite, POCC and PIPS, is proposed to provide higher performance for computationally intense applications.
    - This integrated design flow enables course grained parallelism in nested loop structures to increase the performance of the final design and also provides an autonomous loop carry dependency analysis and further loop optimizations.
    - Efficiency of the proposed system is evaluated on both Vivado HLS simulation environment and Xilinx ZC-702 development board that includes Zynq-7000 series FPGA. Performance of the proposed method is compared with the built-in loop optimization methods of Vivado HLS.
- Although POCC is a powerful polyhedral model-based analysis and optimization tool on its own, it does not include loop skewing transformation. In this thesis, functional correctness of loop skewing is examined and potential advantages of combining of loop skewing and loop tiling are discussed. Loop tiling is a polyhedral loop transformation that POCC is capable of.

- Jacobi 2D iterative stencil kernel is selected and used to verify the performance of skewed-tiling transformations and our novel HLS design flow at same time.

1.4. Thesis Outline

The rest of the thesis is organized as follows:

In chapter 2, background information about MPSoCs architecture is given. Homogeneous and heterogeneous MPSoC systems are described in detail and differences between these systems are reviewed. FPGA based SoC platforms are explained and the way to use these platforms as hardware accelerators are discussed.

In chapter 3, High Level Synthesis (HLS) is explained and step by step HLS-based design flow of FPGA in detail. Also, optimization capabilities of current HLS tools for computationally intense applications and limitations of HLS tools are given.

In chapter 4, information about Polyhedral modelling with its associated crucial mathematical definitions are presented. Usage of the polyhedral model to analyze dependency in nested loop structures is discussed entirely.

In chapter 5, some related works about HLS and polyhedral optimizations are given and differences between these works and this thesis work is also discussed.

In chapter 6, all details of the proposed semi-autonomous approach for computational intense applications are expressed comprehensively.

In chapter 7, an evaluation and the test results of the proposed design flow are given and discussed for different types of loop kernels.

In chapter 8, conclusion and possible future works are given

CHAPTER 2

SOC BASED PLATFORMS AND HARDWARE-SOFTWARE CO-DESIGN

2.1. MPSOC Architectures

In order to adapt to increasing computation demands, basic response of traditional processor in industry has been to raise clock frequencies of processing systems. On the other hand, while clock frequency increases, undesirable problems such as excessive power consumption, other thermal issues, etc. may arise. Therefore, leading integrated circuit (IC) vendors focus on the design and production of embedded devices including many cores without increasing clock frequency. With the development of System-on-Chip (SoC) technology, market leaders have started to fabricate Multiprocessor System-on-Chip (MPSoC) including multiple processing elements (PE), programmable logic (PL), memory systems and I/O components.

Architecture of MPSoCs differ depending on the integrated components, which may have specific role, based on the desired applications. Consequently, there are two main categorizes of MPSoCs, one is homogeneous and the other one is heterogeneous.

2.1.1. Homogeneous MPSoC

Homogeneous MPSoC is an architecture embeds one or more programmable blocks that are all same, in a single IC. This model also known as parallel architecture model. The basic idea behind this architecture is dividing workload of single powerful processor among different physical resources in order to reduce total execution time.

A homogeneous MPSoC, consisting of programmable processing elements, can serve acceleration on desired algorithm or reduced power-consumption by decreasing operational frequency and power supply thanks to its inherent parallel structure.

However, theoretically optimal parallelism cannot be possible for most of the time due to the limitations over flexibility and scalability issues of memory organization, communication infrastructure. A typical homogeneous architecture example can be seen in figure 2.1 [7].

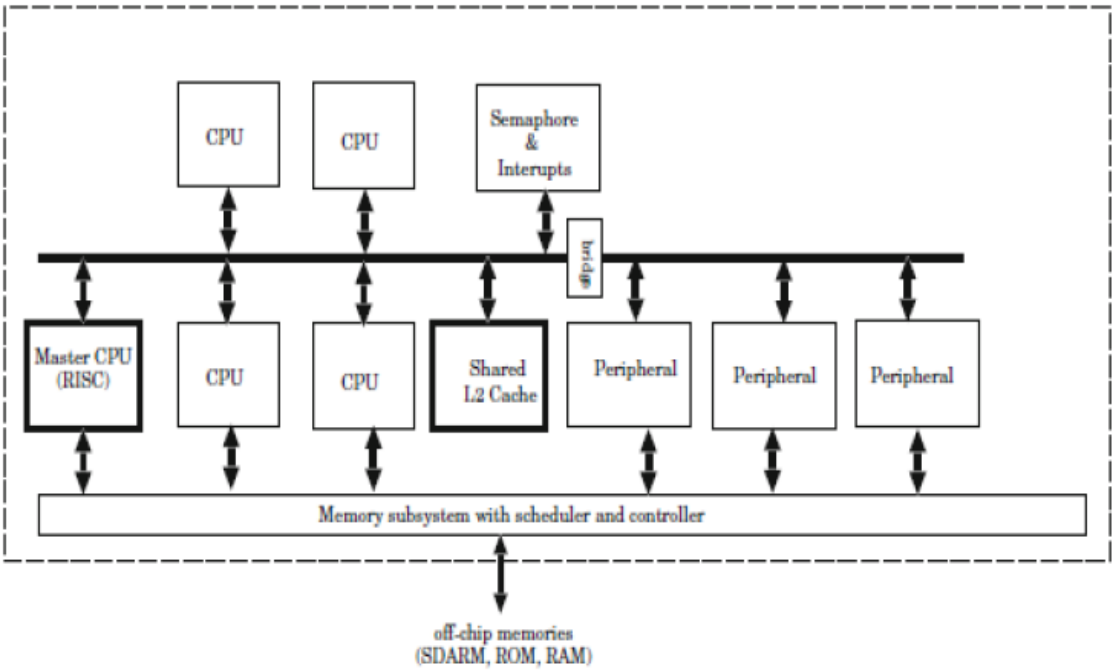


Figure 2.1. Typical Homogeneous MPSoC Architecture Example

2.1.2. Heterogeneous MPSoC

Heterogeneous MPSoC is an architecture, which embeds different types of processing elements such as one or more general purpose processors, FPGAs, DSPs instead of same type of operating blocks in a single IC. This is the main difference between heterogeneous and homogenous architectures.

Heterogeneous MPSoCs serve better flexibility in comparison to homogeneous ones. They may include components, which provide post-fabrication flexibility such as

FPGAs and DSPs. Processing elements that can be seen in various heterogeneous systems, are basically classified as follows:

- Application Specific Integrated Circuit (ASIC)
- Application Specific Instruction Set processor (ASIP)
- General Purpose Processor (GPP)
- Digital Signal Processor (DSP)
- Field Programmable Gate Array (FPGA)

A typical heterogeneous MPSoC example can be seen in figure 2.2 [8]. Test environment, studied in this thesis, is a kind of heterogeneous MPSoC platform, which will be briefly described in section 7.1.

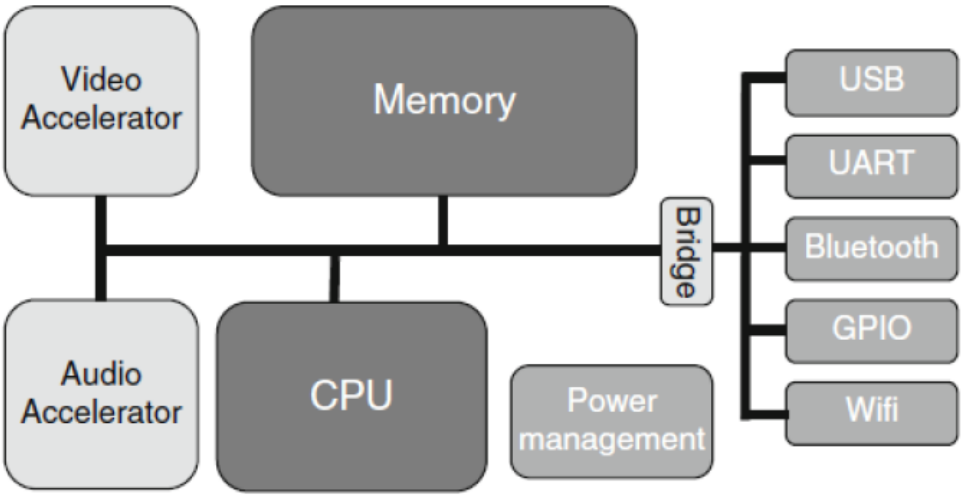


Figure 2.2. Example of Typical Heterogeneous MPSoC Architecture

2.2. SOC FPGAs

FPGA is one of the continuously evolving technologies in today's world. The newest development in FPGA technology is the design and production of System on Chip (SoC) FPGA devices. FPGA-based SoC devices accommodate both hard processor

cores and programmable logic on the same IC. Before the release of SoC-FPGAs, programmable logic architecture was controlled by external processors via connecting them to FPGAs over external interfaces. Merging of FPGAs and CPUs in same IC brings some advantages such as reduction of communication latency, higher bandwidth and smaller physical size. A basic illustration of change in FPGA architecture can be seen in figure 2.3 [9].

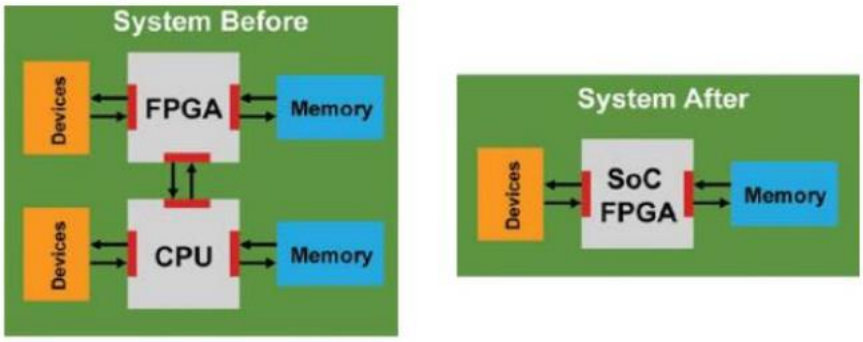


Figure 2.3. FPGA Architecture Change

SoC-FPGAs can be also advantageous for replacing ASICs. SoC-FPGA development costs are much lower and the development requires less amount of design time. Design process is much more flexible because of rewritable firmware at any time. In this thesis, we opted for a SoC FPGA platform from Xilinx Zynq family named as Zynq-7000 as our experimental evaluation and test environment.

2.3. Hardware-Software Co-Design in SoC FPGAs

The concurrent development of hardware and software side of SoC platforms is called as hardware-software co-design. The main idea behind hardware-software co-design

is dividing workload of a system among hardware parts running on FPGAs and software part running on CPUs.

A basic overview of hardware-software co-design flow can be seen in figure 2.4 [10]. Detailed analysis of an expected design in terms of performance, power consumption, design area and other constraints should be the first step of hardware-software co-design process in order to reach to better results in hardware-software partitioning step. Hardware-software partitioning is a procedure that is efficiently split the design blocks among software and hardware parts of SoC systems. After determining which part of the design runs on hardware and which part on software, next step is coding and simulation of each part separately. After completing this step, the design should also be validated via an integrated simulation of hardware and software parts. This procedure is called as co-simulation. After a successful co-simulation, designs could be linked and implemented on development environments.

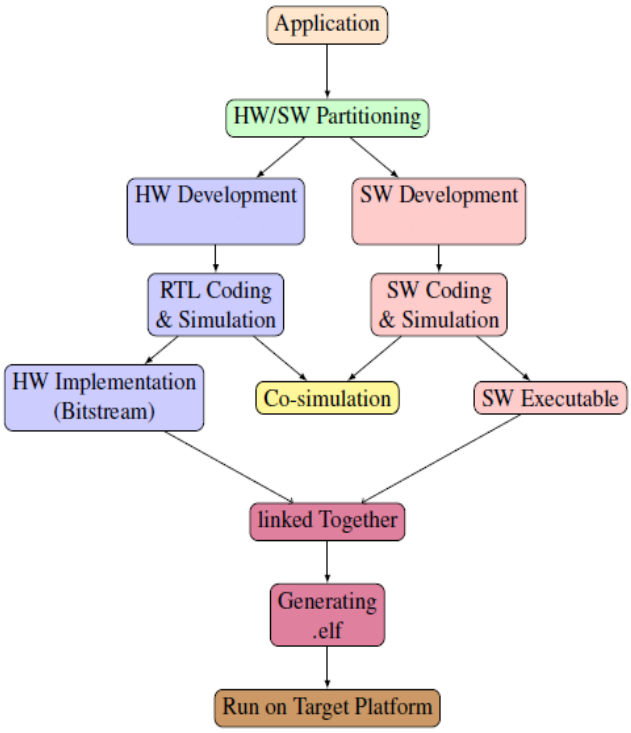


Figure 2.4. Hardware-Software Co-Design Flow Example

2.4. Hardware Acceleration

Hardware acceleration is a method that involves completely or partially implementing an algorithm on dedicated hardware circuit instead of a processor in order to reduce elapsed time while the algorithm runs [12]. Figure 2.5 demonstrates a possible hardware implementation of "If" statement that is repeatedly used on any kind of algorithm [11]. Undoubtedly, these circuits should give the same result if it is compared with their software counterpart.

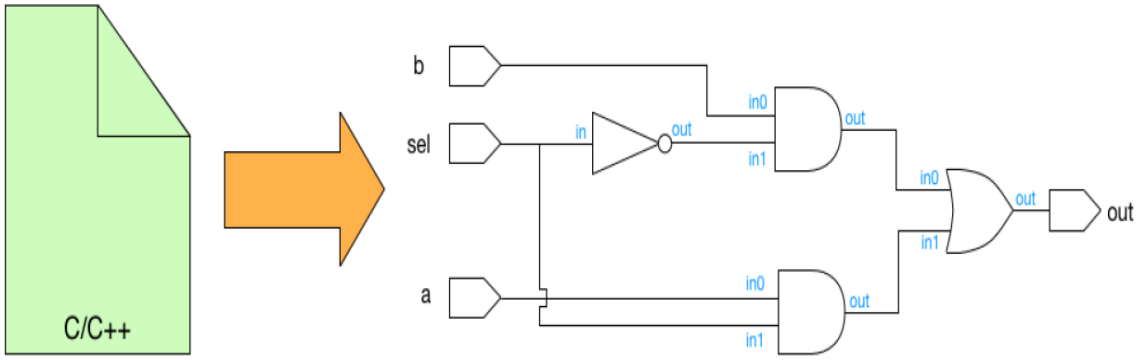


Figure 2.5. Possible Hardware Implementation of If Statement

A hardware designer should entirely understand each hardware block and how these blocks can be integrated in the overall design to reach an efficient solution at the end. Undoubtedly, this is really a time-consuming process that requires well educated hardware designers. Hardware Description Language (HDL) is extensively used in this workflow. HDL based design resembles to assembly language design because it requires low-level abstraction. Consequently, hardware designers cannot skip any signal detail or changing behavior of state machines over time. Besides, debugging in hardware design is extremely complex, requiring knowledge about timing constraints, waveforms, etc.

Although design, validation and implementation steps of hardware design are extremely hard, employment rate of hardware designers is increasing due to need for speeding-up one or more portions of demanding applications, instead of executing sequentially on traditional CPUs. It is usually clear that algorithms implemented in CPUs could not be fast as algorithms running on dedicated hardware because of CPU's sequential structure.

In order to reduce overhead of traditional CPUs, Graphic Processor Units (GPU), consisting of dedicated hardware running special instructions very fast, were also released. Even though GPUs are designed to mainly for graphics computations, its parallel architecture is used naturally for intense scientific computations in recent years. However, main drawbacks of GPU based accelerators are excessive power consumption and thermal problems if compared to FPGA based hardware accelerators. Thus, GPUs could be regarded as sub-optimal for accelerating computationally intense algorithms.

If its power efficiency and flexibility are taken into account, FPGA based hardware accelerator seems to be best alternative for many applications. In recent years, in order to reduce difficulties of the design procedure of FPGAs, high level synthesis tools (HLS), which automatically convert and optimize high level languages to hardware description languages, have been released by FPGA vendors. In this thesis, all designs, running on FPGA, have been created using the HLS tool of Xilinx Company that is called as Vivado HLS.

CHAPTER 3

HIGH LEVEL SYTHESIS

3.1. What is HLS?

High level synthesis (HLS) is an automated design process, which describes behaviors of desired algorithms and defines digital circuits according to this behavior [13]. The ultimate aim of HLS is to raise abstraction to a higher level in hardware design.

Hardware design procedure has changed from defining individual transistor and wires to design of Boolean model of computations using logic gates (and, or, xor...etc.) and flip-flops, to register transfer level (RTL) based design of desired circuits. HLS serves one step beyond of abstraction level served by traditional RTL design. HLS enables designers to entirely focus on functional level designs instead of low-level cycle-by-cycle operations.

Basically, HLS can do the following things automatically to reduce the work-load of a designer in comparison RTL based design:

- HLS figures out the potential concurrency in design.
- HLS can automatically adopt the design to desired frequency.
- HLS can generate interface to connect to the rest of the system easily.
- HLS maps data on designs to storage elements automatically considering a balanced resource usage and bandwidth.
- HLS optimize the design according to user directives or automatically to achieve more efficient implementation.

New generation HLS tools provide different high-level input language options, platform-based modelling and a domain-specific technique [14]. They can offer better

adaptation to industrial needs. Nowadays, demands for high-quality HLS solutions is growing day by day since design period of RTL based systems increase dramatically due to advancement in SoC chip capacities [14].

3.2. C/C++ Based FPGA Design Flow in SOC platforms

In order to completely design a SoC system which use a FPGA as an accelerator for a computational intense part of an algorithm, it is now mandatory to both consider how design the accelerator and how to integrate it to the target platform. HLS tool designs and integration steps can be handled separately.

3.2.1. High Level Synthesis Design Flow

Due to recent advancement in HLS technology, High-Level Synthesis (HLS) tools can be used to design entire FPGA-based accelerator applications [16]. HLS tool flow encloses several steps in order to develop hardware design from input software code to RTL bitstream. HLS tool requires hardware specifications of desired platform in order to make HLS be aware of the physical capacity of the system such as logic structure and available processing and memory resources as shown in figure 3.1 [17]. Because of the flexibility of FPGAs, some constraint defined by the user such as clock speed, degree of parallelism, etc. are also required to be given to HLS as an input.

As illustrated in figure 3.1, the first step of the flow is compilation of source code given as an input to intermediate representation (IR). Then, HLS compiler can generate the data flow graph (DFG) for further steps and optimizations. Now, HLS can determine the high-level hardware structure by using the DFG, information about desired hardware platform and user defined constraints. The quality of generated hardware mostly depends on applied optimizations in this step. Different combinations of compiler optimizations can result in huge performance variations on design outcome.

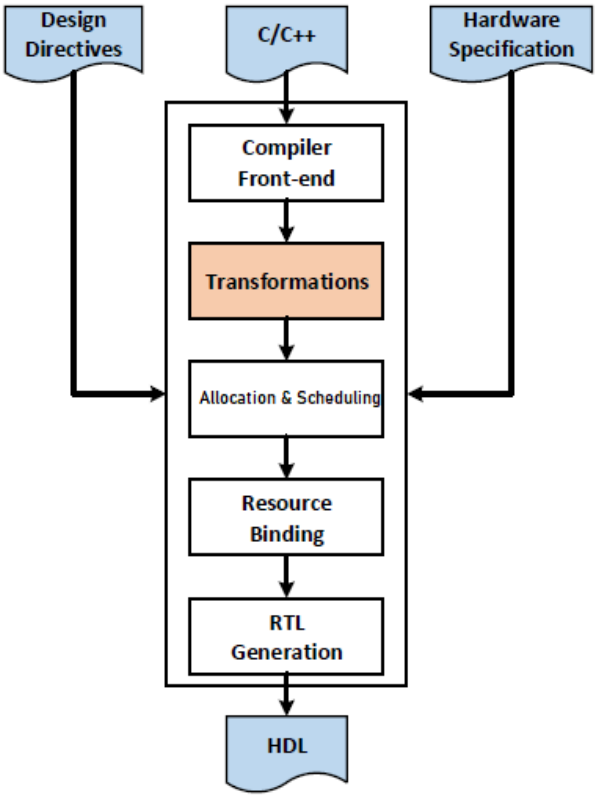


Figure 3.1. Standard HLS Tool Flow

After establishing DFG and desired hardware structure, the following steps are allocation, scheduling and resource binding, which are fundamental concepts can be seen in all types of compilers. In order to grasp these, consider an example code portion in figure 3.2 [18].

```
1  void summation( int a, int b, int c, int d, int & dout )
2  {
3  2: int t1,t2;
4  3: t1 = a + b;
5  4: t2 = t1 + c;
6  5: dout = t2 + d;
7  6: }
```

Figure 3.2. Example Code Portion (dout=a+b+c+d)

The data flow graph of example code can be illustrated as in figure 3.3 [18]. Each node of the DFG represents an operation which is an adder in this example in final design.

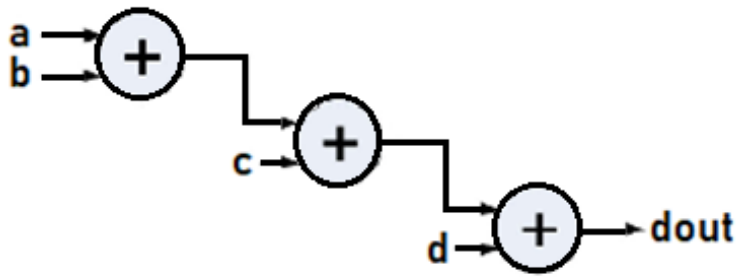


Figure 3.3. Data-Flow-Graph of dout=a+b+c+d

After the construction of DFG, each operation is converted to a hardware resource, which is needed during scheduling. This procedure is called as resource allocation. Allocation procedure determines which hardware resources (e.g., functional blocks, storage components) is required to satisfy the design constraint. These resources are chosen from the RTL component library of the desired FPGA as shown in figure 3.4 [18].

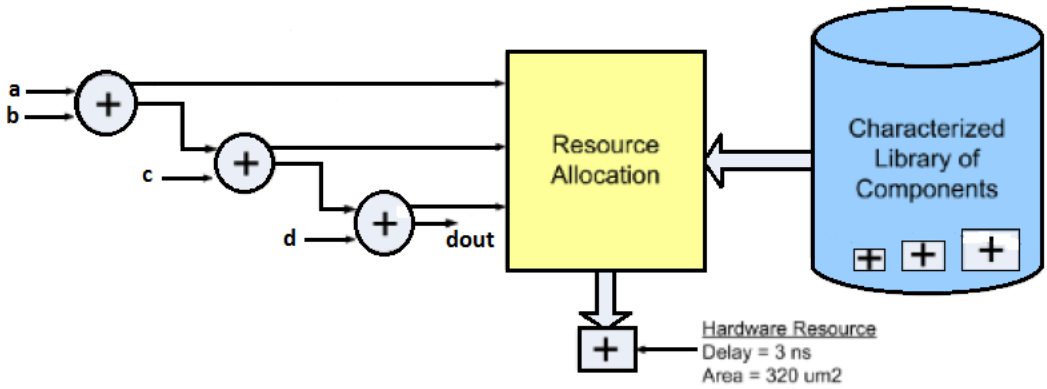


Figure 3.4. Resource Allocation of dout=a+b+c+d

After resource allocation, it is necessary to determine which operation proceed in which clock cycle. This process is defined as scheduling which can be seen in figure 3.5 [18]. Directives specified by users, frequency of the clock cycle, properties of target devices have critical impacts on deciding how operations are scheduled [17].

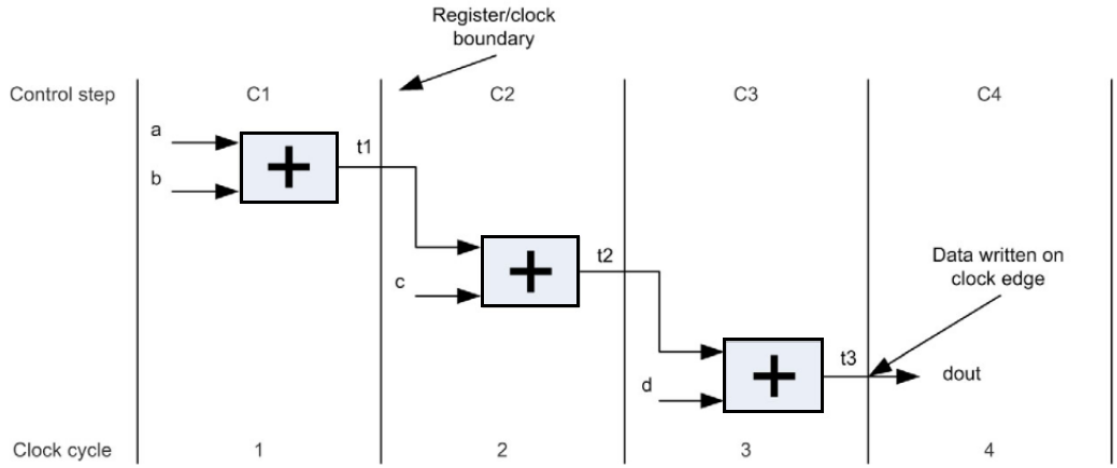


Figure 3.5. Scheduled Design of dout=a+b+c+d

After the scheduling operation, operations, which run on the same clock cycle, can be bound to the same memory unit over a range of non-overlapping or mutually exclusive interfaces [19]. Binding is a procedure that assign hardware resources for operations, which run on same cycle.

Finally, after the binding operation, RTL design corresponding to the given input algorithm is generated.

3.2.2. Integration of HLS Design to SOC platform

Complete design on an FPGA-based SOC platform while taking advantage of the high-level abstraction of HLS tool has three main stages. The first stage is creating the custom RTL code using HLS tool flow described entirely in section 3.2.1. This created

RTL is converted to intellectual property (IP) core to use in the second stage which is called as IP integration. In this stage, IP core from HLS is integrated with the other IPs designed using the traditional RTL flow and processor system. The final stage is creating a software application in the processing system to use the integrated block designs. The tool flow of SoC FPGA, from Xilinx Company, which is used in the entire parts of this thesis, can be seen on figure 3.6. This tool is called as Vivado [17].

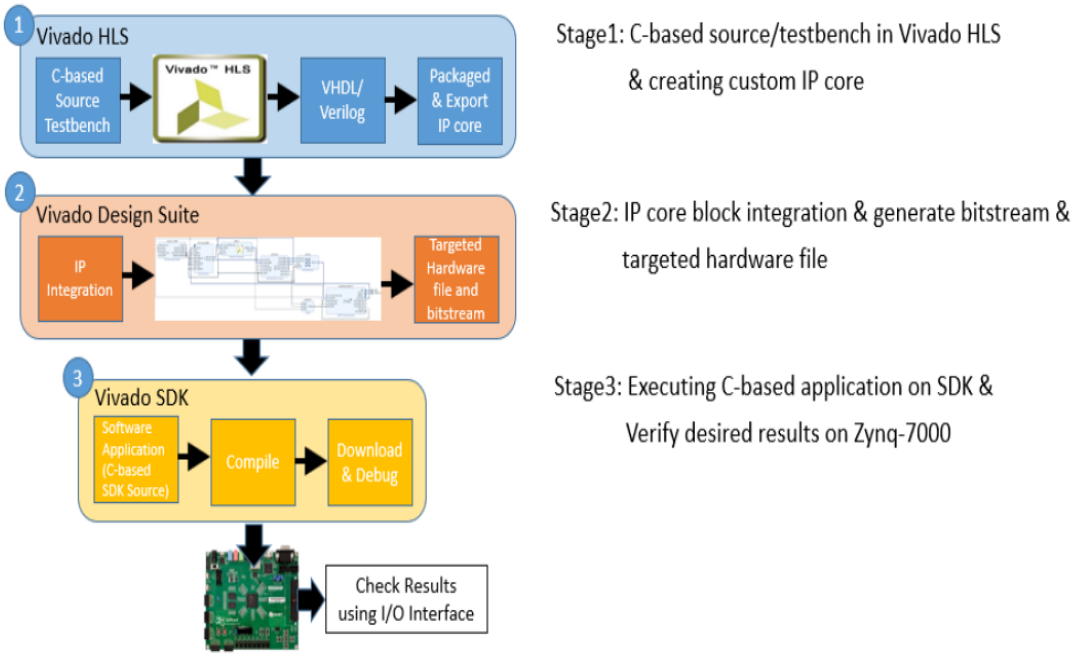


Figure 3.6. HLS design flow in Vivado

3.3. Design Optimizations in HLS

Performance gains (throughput, latency, area usage, etc.) of HLS-based design mostly depend on optimizations applied while converting sequential high-level language-based design to RTL design as stated in section 3.2.1. Commercial and academic HLS tools offer various types of basic optimizations for loop kernels, which are commonly encountered and important parts of computationally intense applications. In this

section, optimization already available for loop kernels in Xilinx Vivado HLS tool are reviewed. These are namely loop unrolling, merging, pipelining and flattening.

3.3.1. Loop Unrolling

Loop unrolling provides execution of consecutive iterations of a loop in a single body in order to divide the loop overhead to a factor of unrolling and also provides reuse of consecutive values that appear multiple times in the loop body. Let's consider the example code portion in figure 3.7.

```
do i = 2, n-1
 a[i] = a[i] + a[i-1] * a[i+1]
end do
```

Figure 3.7. Example Code Portion for Loop Unrolling

The following loop shown in figure 3.8 is the resultant code of code in figure 3.7 after loop unrolling with factor 2. In this example, there is a data dependency between a[i] and a[i+1] thus, all operations in the loop body cannot be parallelized.

```
do i = 1, n-2, 2
 a[i] = a[i] + a[i-1] * a[i+1]
 a[i+1] = a[i+1] + a[i] * a[i+2]
end do
```

Figure 3.8. Unrolled Code Example

3.3.2. Loop Merging

Loop merging is to combine different loop operations in a single loop body. If multiple loops, having the same bounds, execute consecutively and there are not any kind of

dependency between the loop contents, application of loop merging is possible. This optimization brings a reduction in the loop's iterations overhead, which is caused by entering and exiting a loop [17]. A typical example of this optimization can be seen in figures 3.9 and 3.10. This optimization is also called as loop fusion.

```
do all i = 1, n
 a[i] = a[i] + c
end do all
do i = 1, n
 x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

Figure 3.9. Code portion before merging

```
do i = 1, n
 a[i] = a[i] + c
 x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

Figure 3.10. Code portion after merging

3.3.3. Loop Flattening

In section 3.3.2, we mentioned about loop merging in order to reduce the latency overhead due to entering and exiting a loop. Sometimes the latency cannot be improved too much for example if two separate basic loops are merged. However, when loops are nested, this improvement may be at dramatically high level. Imagine a nested loop with 100 iterations in outer loop and 5 iterations in inner loop. This would result in 200 cycles overhead to enter and exit to inner loop. Loop flattening could eliminate this issue by totally unrolling the inner loop [17].

3.3.4. Loop Pipelining

HLS tool provides two different types of pipelining for loop kernels, which are loop pipelining and loop dataflow. Loop pipelining is applied on single loop body to

pipeline sequential instructions in the loop body; however, loop dataflow pipelines a series of loops.

Loop pipelining is implemented on loop body in order to make all instructions pipelined. For instance, it is assumed that loop body include three sequential operations: read, write and computation, it could be pipelined to execute read operation on every clock cycle instead of one of third cycle as shown in figure 3.11 [17]. Hence, each loop iteration begins on every clock cycle before the previous one completes. In order to apply loop pipelining, it is required that no data dependency between two consecutive iterations exist.

Dataflow is used on bigger scope. This optimization pipelines different loops having no dependency among them. This optimization also works for a series of functions in the same manner. The biggest constraint of dataflow is that variables must be generated and consumed in single loop/function body as seen in figure 3.12 [17].

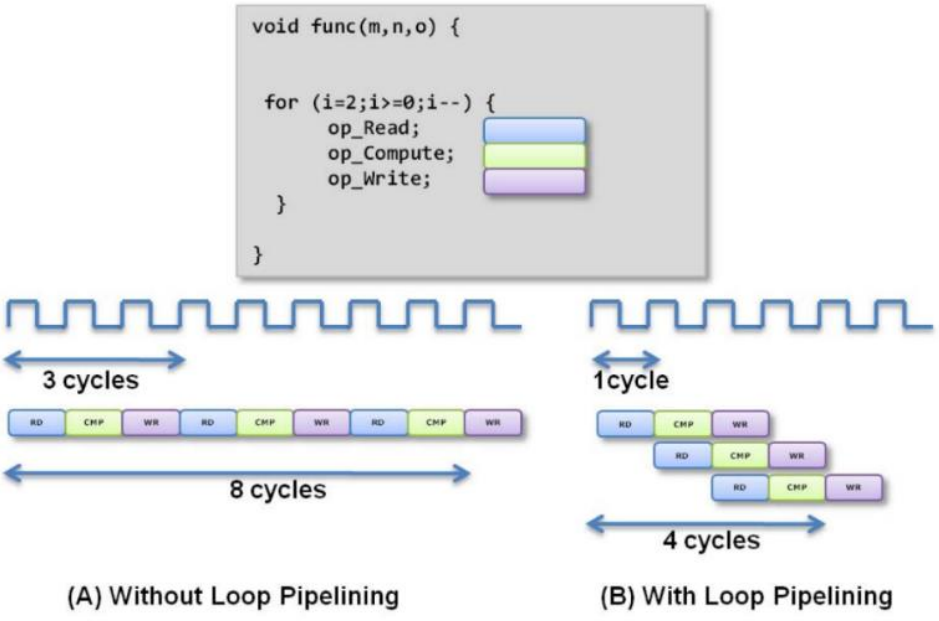


Figure 3.11. Loop Pipelining Example

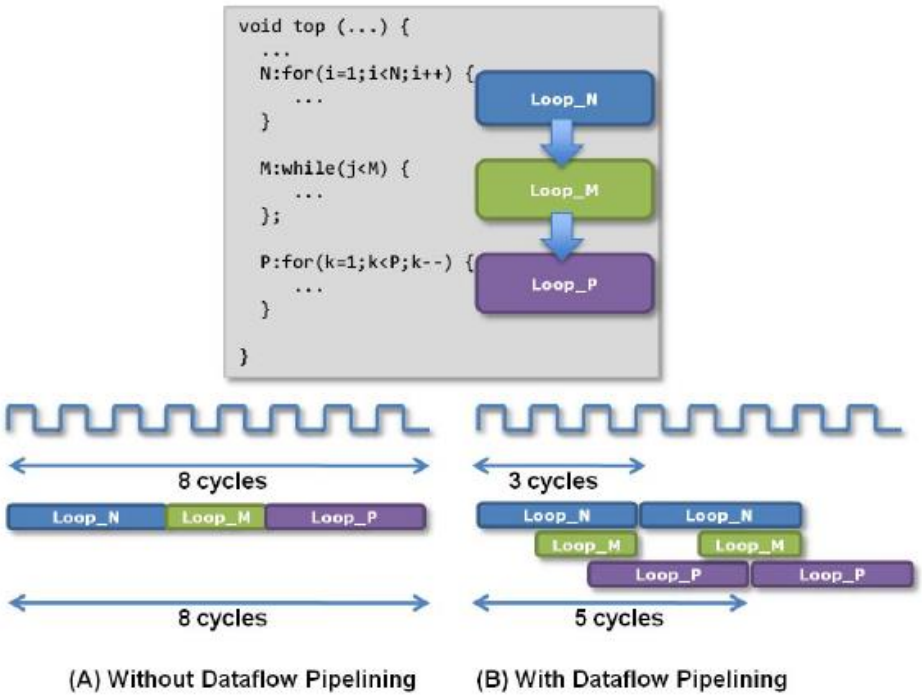


Figure 3.12. Loop Dataflow Pipelining

3.4. Limitations of HLS

Recent studies about HLS have shown that performance of FPGA-based systems which are designed by using HLS tool flow are as competitive as hand-crafted RTL design [6]. Because of its high-end design productivity, HLS seems to better choice instead of RTL design, especially when having limited time for developing an accelerator system. On the contrary, in practical view, it is necessary to spend noticeable amount of time on rewriting code to make it more HLS friendly and apply tuning using HLS directives to get high design quality [20]. In this section, two major limitations of commercial HLS tools are presented, which requires considerable research effort to handle in order to improve HLS technology.

One of the main challenges of HLS is accurate and efficient exploration of the design space. In the literature, there are various approaches for design space exploration (DSE) to generate convolution neural network (CNN) accelerators using HLS. Zhang

et al. offer a roofline model for its design space exploration, which includes implementation of different types of loop transformations [21]. Also, Suda et al. suggest a DSE method for CNN to increase the design throughput [22]. It is also crucial to think about computing characteristics beyond performance and resource usage. Gao et al. generate a method to handle complicated trade-off between area, latency and error of floating-point programs [23]. Exhaustive DSE can be very hard and time consuming and needs application-specific information. Thus, there is no accepted standard for DSE problem and it is still an open research area.

Although various HLS tools offer different kinds of optimizations, these optimizations cannot work properly on all kinds of software. For example, dynamic memory allocation and recursion are commonly used in software programs, however there is no support for these in many HLS tools. Winterstein et al. offer a logic separation and program analysis technique to handle dynamic data structures [24]. Even though, there are numerous researches on this problem, rewriting of software code is still required for HLS design. Moreover, loop optimization techniques of HLS tool such as Vivado works only on instruction in loop bodies and perfectly parallelizable instance of different functions. When there is any kind of irregularity in the source code, parallelization of nested loop structures may be impossible although it is theoretically possible. The absence of optimizations for these unavoidable irregularities is one of the major limitations of current HLS tools. In this thesis, we therefore offer a technique to reduce the effect of these limitations, in order to increase the available course grained parallelism.

CHAPTER 4

POLYHEDRAL MODELLING FRAMEWORK

4.1. Motivation

In order to employ parallelism for loop kernels in HLS, we propose in this thesis to perform optimizations based on polyhedral modelling. Basically, the polyhedral model is an algebraic representation of loop kernels. Use of mathematical modelling in loop kernels was proposed in early years of modern computing systems [36]. Popularity of polyhedral model for computational optimizations has increased dramatically in academic researches with the advancement of polyhedral tools. They cannot only cover automatic parallelization but also memory management and data locality optimizations. In this chapter, the theoretical background of polyhedral modelling is reviewed briefly.

4.2. Mathematical Background of Polyhedral Model

This section provides mathematical definitions for polyhedral modelling. More detailed information about polyhedral modelling can be found in [37].

Let's cover some definitions:

Affine Function:

A function $f: K^m \rightarrow K^n$ is affine if there exists a vector $\overrightarrow{b} \in K^n$ and a matrix $A \in K^{n*m}$ such that:

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

Affine Space:

Affine space is a set of vectors that is closed under affine combination.

Affine Half Space:

Affine half space of K^m is described as the set of points:

$$\{ \vec{x} \in K^m \mid A\vec{x} \leq \vec{b} \}$$

Affine Hyperplane:

Affine hyperplane is a $n-1$ dimensional affine subspace of an n dimensional space. Hyperplane separates the space into two half-spaces which are called as positive and negative half-space as shown in figure 4.1. Each half-space can be described by an affine inequality. Representation of hyperplane for 2D and 3D space could be seen in figure 4.1 [38].

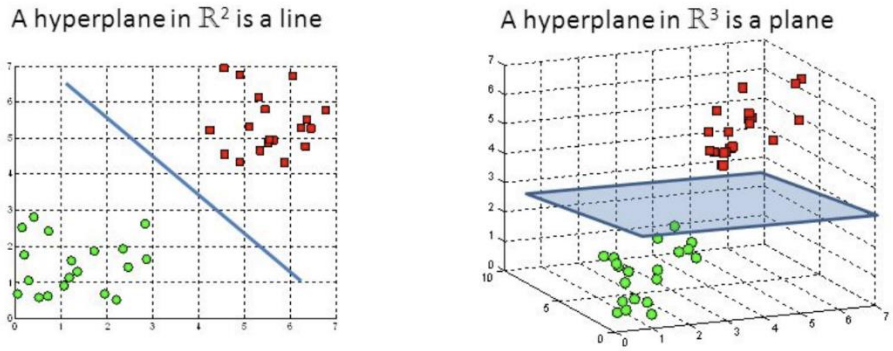


Figure 4.1. A Hyperplane in 2D and 3D Space

Polyhedron:

Polyhedron is a set $S \in Q^n$ which satisfies the following inequality and which can also be defined as an intersection of closed affine half-spaces.

$$P: \{\vec{x} \in Q^n \mid A\vec{x} \geq \vec{b}\}$$

Polyhedral Statement:

Polyhedral statement is a set of program instructions that ensure the following conditions:

- If program instruction is a conditional assignment with an affine condition, it is not a polyhedral statement.
- If program instruction is a loop statement with affine bounds, it is not a polyhedral statement
- Program instruction must have only affine subscript expressions for array accesses in order to be a polyhedral statement.
- Program instruction should not generate flow-control effect in order to be a polyhedral statement.

Iteration Vector:

Iteration vector of an m dimensional loop nest is a vector consisting of iteration variables, $\vec{\imath} = (i_0, i_1, \dots\dots\dots, i_{m-1})$, where $i_0, i_1, \dots\dots\dots, i_{m-1}$ are iteration variables from outermost to innermost loop.

Iteration Domain:

Iteration domain covers all possible values surrounding loop iterators through a set of affine inequalities.

The iteration domain (ID) $D \subseteq Q^m$ is the iteration vector set of nested loops, and are described by a set of linear inequalities [39].

$$D = \{\vec{\imath} \mid P\vec{\imath} \geq \vec{b}\}$$

Each point inside the polyhedron corresponds to one execution of a statements inside the loop body as shown in the nested loop example in figure 4.2 [39]. This model provides coordination of values of loop iterators and execution of statements in the

loop body and gives an opportunity to compilers to manipulate statement execution and iteration orders.

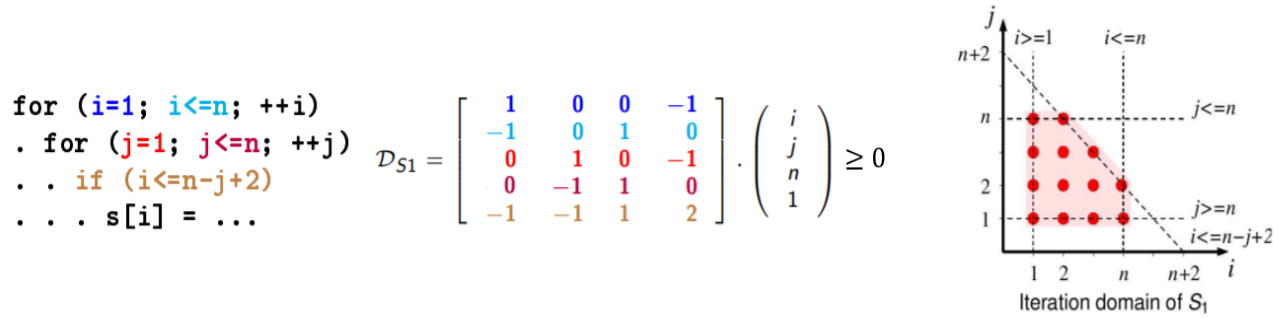


Figure 4.2. Iteration Domain Example

Order of Execution:

Let's consider loop body with statement W. Evaluation of loop body on $\vec{x}$ iterator is defined as an operation and denoted as $< W, \vec{x} >$, where $\vec{x} \in D(W)$. Execution order of all operations of all statements in a loop body is called as schedule. In literature, schedule is also called as lexicographic order. The formal definition of lexicographic order is as follows [39]:

Let's assume two operations $< W, \vec{x} > and \ < R, \vec{y} >$ and m dimensional iteration domain

$$< W, \vec{x} > \succ < R, \vec{y} > \leftrightarrow (x_0 > y_0) \vee (x_0 = y_0 \wedge x_1 > y_1) \ \vee \ (x_0 = y_0 \wedge$$

$$x_1 = y_1 \wedge \ x_2 > y_2 \ ) \vee \ldots \ldots \vee (\ x_0 = y_0 \wedge \ \ldots . \wedge \ x_{m-2} = y_{m-2} \wedge x_m > y_m)$$

4.3. Data Dependencies

Mathematical modelling of data dependencies is essential for the efficiency of the polyhedral model because all programs in which polyhedral transformations are used, should give the same result with the original version. If the dependency is preserved,

this condition is automatically satisfied. In this section, some crucial definitions for data dependency representation and analysis are given.

Subscript Function

Let's assume that the set of arrays A_n, a reference point in array $B \in A_n$ and a statement in loop body S, are given.

S can be written as $< B, f >$ where f is a subscript function [40].

Data Distance Vector

Let's assume that two different functions f_A^R and f_A^W exist and α and β are two different iterations of the innermost loop of a nested loop. Data distance vector is described as:

$$\delta\,(\alpha,\beta)_{f_A^R f_A^W} = f_A^R(\alpha) - f_A^W(\beta)$$

Lexicographically Non-Negative Distance Vector

If the left-most non-zero entry of the distance vector is positive or all elements of this vector are zero, this distance vector is called as non-negative distance vector.

Legal Distance Vector

A distance vector is legal, if it is lexicographically non-negative (non-positive) and loop indices increase (decrease).

Bernstein Conditions

Assume that two different statements are given, it can be said that there exists a kind of dependency between them, if following conditions hold [40]:

- They refer to the same memory location

- One of the statements is a write operation
- Both statements are executed

When the conditions mentioned above are satisfied, three different types of dependencies can be discussed [41]:

Read After Write (RAW)

A Read-After-Write dependency occurs if an instruction requires the result of a previously executed instruction. This dependency is also called as true dependency or flow dependency. If pseudocode in figure 4.3 is examined, statement in line-2 requires execution of line-1 and also consecutively, statement in line-3 requires execution of line-2.

```
1  A=5;
2  B=A;
3  C=B;
```

Figure 4.3. RAW Dependency Pseudocode

It is impossible to run these three instructions in parallel. In other words, instruction level parallelism is not an option for this pseudocode because of the dependency between each instruction.

Write After Read (WAR)

A Write-After-Read dependency occurs if an instruction writes to a memory location which has not been read by a previous instruction. This dependency is alternatively called as anti-dependency. Anti-dependency is a good example for name-dependency.

In order to remove name-dependency, only renaming of the variable is enough. However, flow-dependency might arise as a result as shown on example code in figure 4.4. Flow dependency occurs between statements in line-2 and line-3.

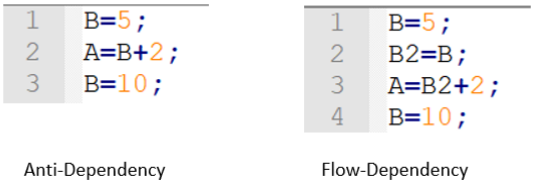


Figure 4.4. Anti-Dependency vs Flow-Dependency

Write After Write (WAW)

A Write-After-Write dependency occurs if an instruction writes to a memory location which a previous instruction wrote. This dependency is alternatively called as output dependency.

Output dependency is a sort of name dependency as the anti-dependency case. That is, they can be handled through renaming of variables. However, renaming variables result in flow-dependency similar to anti-dependency as shown in figure 4.5.

Figure 4.5. Output-Dependency vs Flow-Dependency

Previously, there was an output-dependency between statements in line-1 and line-3. After renaming variables in order to remove output-dependency, flow-dependency occurred between line-1 and line-2 at the example code in figure 4.5.

Unfortunately, in real-word applications, computational algorithms are much more complex. Hence, they are not limited to simple sequential instructions. These algorithms usually consist of variables in deep nested loops with complex

dependencies. Thus, it is required to analyze mutual dependencies between different iterations of nested loops. At this point, loop-carried dependency concept should be introduced.

Loop-Carried Dependencies:

Loops are the way of executing the same sequence iterations with different data automatically. One of the questions we try to answer in this thesis is "Can two different loop iterations execute in parallel or is there any data dependency between them?"

Let's consider example loop in figure 4.6. It is easy to observe and say that this loop is completely parallelizable. Since, any loop iteration depends its own. Changing the order of calculations does not change the result. Relaxation of execution order provides executions of this loop on parallel hardware platforms.

```
1  for (i = 0; i <= N i++)
2  {
3   A[i]= A[i]+ B[i];
4  }
```

Figure 4.6. Completely Parallelizable Loop

Definitely, loop in figure 4.6 is ideal for parallel processing, on the other hand, nearly all of the loops in computationally intense applications include different iterations which depend another iteration.

Variation of loop-carried dependencies are similar to dependencies for sequential instructions which were defined before. The only difference is that dependencies are now between loop iterations instead of sequential statements.

When the loop in figure 4.7 is examined, it is observed that the loop is similar to the one in figure 4.6, the difference being in array indexing in line-3. This difference results in loop-carried flow dependencies.

```
1  for (i = 0; i <= N i++)
2  {
3   A[i]= A[i-1]+ B[i];
4  }
```

Figure 4.7. Loop-Carried Flow Dependencies Example

It is useful to manually unroll several iterations of the loop in figure 4.7 to examine this dependency issue. Unrolled iterations in figure 4.8 illustrates that second statement depends on the result of first one, and third statement depends on the second one consecutively. This kind of flow-dependency could be seen in a wide range of algorithms used in different applications.

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

Figure 4.8. Unrolled Example Code

The formal definition of loop carried dependency as follows:

- Assume a nested loop having two or more legal distance vectors:
  $(D_1, D_{2,} \ldots., D_m)$, $m \geq 2$
  Each distance vector of n dimensional nested loop could be illustrated as:
  $D_k = (d_1, d_2, \ldots., d_n)$,
  Dependency is carried at loop level i if d_i is the first non-zero element of D_k.
  Each execution of a loop could be executed in parallel when it carries no dependencies.

This formal definition is crucial in order to determine whether or not a nested loop is parallelizable at any loop level using the distance vector concept.

In some cases, it is impossible to execute such a loop in parallel. In other words, eliminating flow dependencies are impossible. On the other hand, for different scenarios, flow dependencies could be handled by loop-transformation techniques which are explained in chapter 6.

4.4. Application Domain of Polyhedral Model

The category of problems, which the polyhedral model targets, are scientific applications having computational workloads in general. The focus of polyhedral analysis are workloads for which all the information is known at compile time. Polyhedral model-based transformations on a source code can only be done by analyzing code statically. Due to the fact that most computationally intense algorithms share this characteristic, they can be analyzed using polyhedral model effectively.

All in all, scope of interest of polyhedral model in this thesis could be re-defined as:

- Static
- Affine
- Pure

Let's examine the conditions required to be satisfied by a code to be static, affine and pure.

4.4.1. Static

A code portion can be defined as static when:

- Loop bounds are known at compile time
- There are no data dependent condition

Example code in figure 4.9 is not static. Since conditional statement in line-5 breaks second condition in order to be static.

```
1  int A[];
2
3  for (i = 0; i<=10; i++) {
4      for (j = 0; j<=10; j++){
5          if (i< A[i])
6              A[i]= i+j;
7      }
8  }
```

Figure 4.9. Non-Static Code Example

4.4.2. Affinity

A code portion could be defined as affine when array indices in loop body are constant or a linear combination of enclosing loops. To illustrate, code in figure 4.9 is affine although it is not static.

4.4.3. Pureness

A code portion could be defined as pure if:

- It does not include global data
- Result does not depend on variables which are not known by the compiler.
- No read or write operations without the compiler knowledge exist.

Thus, pass value by reference is not allowed in order to generate pure code. An example non-pure code and its pure counterpart can be seen of figure 4.10. The only difference between them is that in line-9, function is called by reference (value) in non-pure (pure) code.

Pure functions are ideal to map on parallel hardware platforms since it is not required to use global variables to exchange data. In fact, efficient implementation of global variables on hardware platforms needs a mechanism that provides some sort of synchronization. This brings a waste of hardware resources and introduces wait states for all hardware blocks in the design.

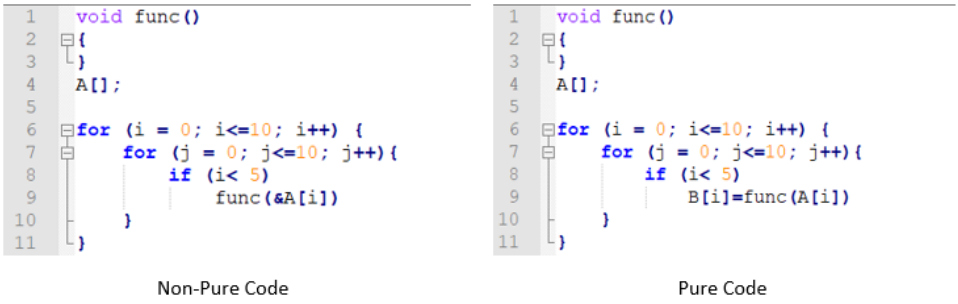


Figure 4.10. Non-Pure and Pure Code Example

4.4.4. Static Control Parts (SCoP)

Static Control Parts (SCoP) are that subclass of nested loops, which can be represented by the polyhedral model [43]. These loops are widely used and results in a performance bottleneck in a high-performance computing (HPC) applications. Array accesses in SCoP statements are same as affine functions, which are previously defined in section 4.4.2. Because of the affine nature of the loops, loop analysis and optimizations such as dependency analysis or parallelization can be applied precisely. Also, SCoPs should be static and pure in order to parallelize them in hardware platforms.

4.5. Usage of Polyhedral Model for HLS-Based FPGA Accelerators

Because of the suitability of FPGA logic and memory elements for parallel computing, FPGA-based hardware accelerators have been widely used in applications such as video/image processing, which demand powerful hardware resources for computation. The most time-consuming part of the computational kernel in these applications are nested loops that can be modeled by using polyhedral model. Polyhedral optimizations can be very appropriate in designing their hardware accelerators because of the computational or memory intense nature of them. In recent years, polyhedral model

has been studied to optimize wide aspects of hardware design such as memory reuse [44], design space optimizations for low power SoC [21].

If FPGAs are designed using HLS, previously mentioned approaches for polyhedral model and FPGA design may be combined to cover applications which are difficult or time consuming to be designed in a hardware platform directly. Thus, we propose and demonstrate to use the polyhedral model-based optimization techniques in hardware accelerator designs and optimizations with the help of HLS tool in order to reduce runtime latency via speeding-up loop executions.

CHAPTER 5

LITERATURE REVIEW AND RELATED WORKS

5.1. High level Synthesis (HLS)

Although HLS tools provide automated conversion from high-level languages to register transfer level (RTL) implementation in order to reduce design efforts and duration, many studies in literature demonstrate that there is a huge performance gap between manual RTL design and HLS-based design for complicated applications. To illustrate, in the study of Liang et al. [51], performance difference between traditional RTL design and HLS-based design of the same algorithm is as high as 40 times. In this study, high-definitions stereo matching application is selected as a benchmark to illustrate the performance gap between the two different design procedures. Hence, different academic works proposed various solutions to address this issue. Ziegler et al. [52] proposed a method of compiler analyses which could guide to map sequential C program into a pipelined implementation for FPGA. Rodirgues et al. [53] offered a technique of execution to accelerate successive tasks having data-dependencies on reconfigurable platforms. Meanwhile, Lie et al. [54] introduce a novel customized optimization based on index set splitting to decrease initiation overhead of pipelined loops to reduce total latency. More recently, Cong et al. [56] have demonstrated how the quality of generated RTL design depends on source level and intermediate level optimizations. They have implemented 56 different optimization techniques and show that some of them have significant importance on hardware quality. Huang et al. [55] has studied the effects of various compiler optimization techniques on circuit generated with HLS. According to study of Huang et al., there are two important factors, namely optimizing methods and their order to improve performance of generated circuits. In this work, six different optimizations methods are implemented on benchmarks and a performance improvement is approximately 30% is achieved.

Even though, there are many researches in the literature that demonstrate how to optimize HLS-based circuit design, there is not any source-to-source compilation tool for HLS-based designs that has capability to analyze and optimize the given source code automatically for efficient circuit generation in computationally intense applications. In this thesis, a novel approach to semi-autonomously analyze and optimize HLS source code to improve the performance of designed FPGA blocks.

5.2. Polyhedral Modelling

Polyhedral modeling is an algebraic representation of loop kernels which is a milestone for computationally intense applications. Widespread utilization area of polyhedral modelling in literature is the design automation and optimization for data reuse in improving memory and cache performance. Meeus et al. [57] shows that although most of HLS tools generate excellent RTL designs, they cannot optimize memory accesses across iteration of nested loops. Unfortunately, overhead of memory access is well-known to be reason for limited circuit performance. Some researchers proposed memory optimizations at the algorithm level. In [58], Cong et al. proposed a method for fitting loop nest to available local storage by using cycling reuse buffer and validated this method using HLS. However, these researches lack an analytic model that provides analysis framework for large and computationally intense kernels.

In more recent works, polyhedral modelling and optimization techniques have been combined with HLS to optimize input algorithms for improving data locality. Bayliss et al. [59] used polyhedral model to design an address generator which increases data reuse. On the other hand, in this work, no loop transformation is taken into account to get better performance. In [60], Jiang et al. proposed an automatic tiling for 1D arrays to optimize memory access schedule. Meanwhile, Wang et al. [61] offer tiling for multidimensional arrays to increase efficiency of stencil computations with polyhedral-based block partitioning. Also, Alias et al. [62] and Pouchet et al. [44]

discussed different memory accesses, data reuse optimization techniques based on loop tiling.

Although, loop tiling is extensively studied to design cache friendly optimized systems in order to reduce memory access overhead using HLS, it could also be used to parallelize computationally intense applications. In this thesis, tiling hyperplane loop transformation methods of polyhedral framework is used to enable course grained parallelization of nested loop structures instead of reducing the overhead of memory access and a semi-autonomous approach based on loop tiling with some further improvements is offered as a starting point towards automatic source-to-source algorithm parallelization tool.

CHAPTER 6

A SEMI-AUTONOMOUS ACCELERATOR FOR COMPUTATIONALLY INTENSE APPLICATIONS

6.1. Motivation

Software programs including algorithms having high computational intensity such as signal processing, deep learning, and multimedia applications contain excessive use of nested loops. Requirement of a kind of accelerated implementation for these programs cannot be ignored. Hardware accelerations designed by HLS tool flow is a good candidate for these implementations. As stated in chapter 3, one of the main problems of HLS tools is its inefficient handling of nested loop structures to reach a minimum execution time. Although, commercial or academic HLS tools offer loop unrolling and pipelining, most of the time, reaching efficient final design cannot be possible due to increasing complexity of algorithms. In literature, there are a lot of loop transformation methods such as loop skewing, loop tiling, loop interchange to increase the efficiency of hardware implementation of nested loop kernels. Even though, these polyhedral model-based loop transformations have been known for many years, they are still not provided by commercial HLS compilers.

All in all, in this chapter, we propose and demonstrate a semi-autonomous system which is a combination of different optimization tools known in the literature to be applied before performing HLS optimization in for loop-kernels for parallel processing of them in FPGA hardware platform.

6.2. Proposed Architecture Overview

The proposed architecture in this thesis, is built in three main different parts, which work in squence. In order to analyze and transform nested loop structures using polyhedral modeling, firstly, it is necessary to detect the static control parts (SCoP) in

given source code as was mentioned in chapter 4. Thus, the first step of our architecture is automatically extracting SCoP modules, which simply are nested loops in the given source code. In the second step, polyhedral representation, which is also known as intermediate representation (IR) of the given nested loops are created then, if it is necessary, suitable polyhedral optimizations are applied on these nested structures in order to create parallelizable loop structures. Finally, the rest of work is to create bitstream to program the hardware platform to use it as a hardware accelerator. In this step, Xilinx HLS tool flow that is reviewed in chapter 3 is used to program the programmable logic (PL) side of Zynq SoC platform. The basic flow diagram of the proposed architecture can be seen in figure 6.1. Each step in figure 6.1 that are to be applied before HLS tool are described in following sections.

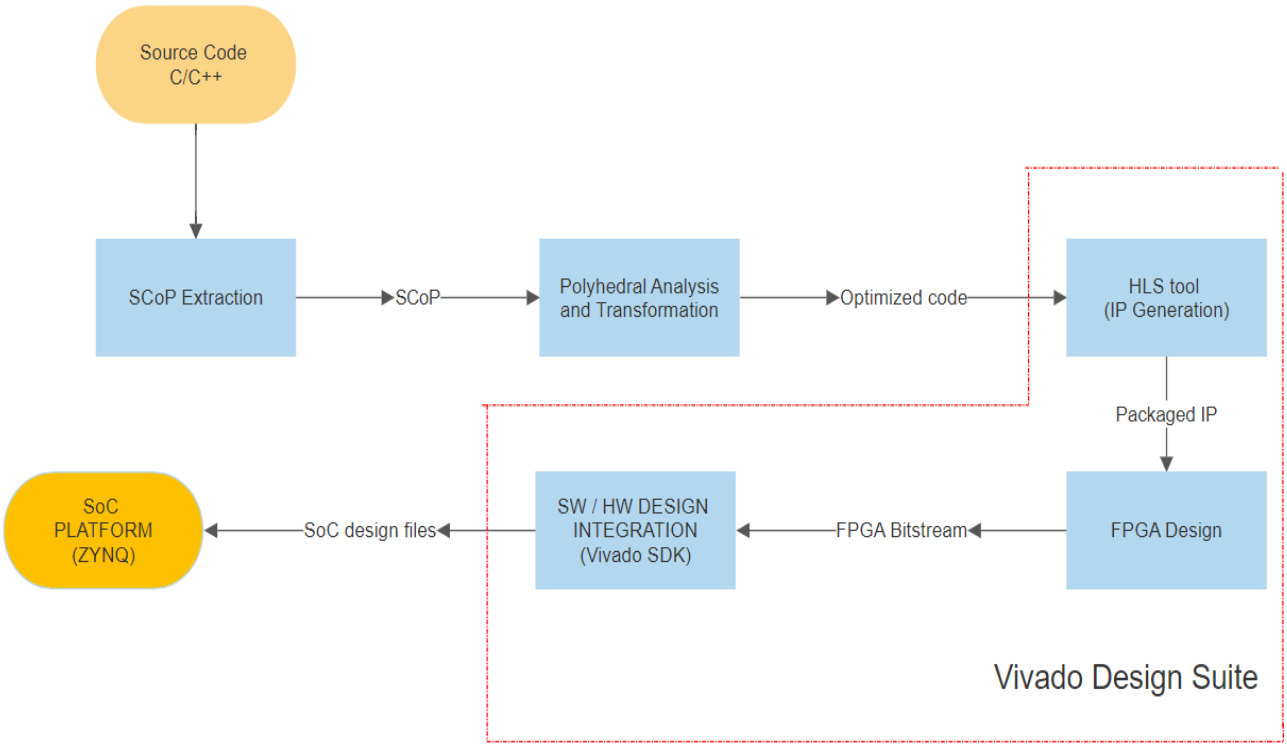


Figure 6.1. Flow Diagram of Proposed Architecture

6.2.1. SCoP Extraction

In our proposel, SCoP extraction is a front-end task to be applied before polyhedral analysis. Most polyhedral analysis tools require pragmas such as “scop” and “endscop” to grasp SCoPs in source code as shown in the example code portion, which is“gemver.c” from Polybench suite, in figure 6.2 [25].

```
int main(int argc, char** argv)
{
  int i, j;
  int n = N;

  /* Initialize array. */
  init_array();

  /* Start timer. */
  polybench_start_instruments;

#pragma scop
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      x[i] = x[i] + beta * A[j][i] * y[j];

  for (i = 0; i < N; i++)
    x[i] = x[i] + z[i];

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      w[i] = w[i] +  alpha * A[i][j] * x[j];
#pragma endscop

  /* Stop and print timer. */
  polybench_stop_instruments;
  polybench_print_instruments;

  print_array(argc, argv);

  return 0;
}
```

SCoP

Figure 6.2. Code example for SCoP pragma

In this thesis work, in order to automatically extract SCoPs from the source code, PIPS [26] is used. Although, PIPS is a source-to-source compiler, in this thesis, we have

only used its “prettyprinter” feature because there is no comprehensive polyhedral model-based loop analysis or optimization method implemented in PIPS compiler.

“Prettyprinter” pass is used for typing pragmas, which define SCoP of the code. Generated code can then be an input for any compiler requiring pragmas for SCoP definition. Figure 6.3 demonstrates required instruction pass for pragmas printing in “gemver.c”.

```
11  alias pocc_prettyprinter 'gemver.c'
12
13  pocc_prettyprinter      >    MODULE.code
14
15      < PROGRAM.entities
16      < MODULE.code
17      < MODULE.static control
```

Figure 6.3. PIPS Pass for “gemver.c”

6.2.2. Polyhedral Analysis and Transformations

As declared in [27, 28], the polyhedral model has reached enough maturity to develop compilers. Authors in [29] offers a set of tools to work on intermediate representation of polyhedral model. Thanks to these tools, polyhedral model can deal with analyzing and optimizing nested loop structures.

In the literature, there are many works that use these polyhedral tools. The most admissible ones are Polly [30], Loopo [31] and Polyhedral Compiler Collection (PoCC) [32]. In this thesis, we have used PoCC for polyhedral model-based analysis and optimizations of nested loop structures to form the autonomous second step of our proposed architecture.

PoCC is a source-to-source compiler that uses polyhedral model for efficient analysis, optimization, and parallelism. PoCC [32] has been built by combining different tools that are already available in literature.

These tools are:

- Chunky Loop Analyzer (CLAN), is used to extract an intermediate polyhedral representation from high-level source code written in C, C++ [32]. CLAN requires enclosed code portions by “pragma scop” and “pragma endscop”. These pragmas are automatically written by PIPS in source as was mentioned in section 5.2.1.
- Chunky Analyzer for Dependencies in Loops (CANDLY), is used to find all polyhedral dependencies from polyhedral intermediate representation that is feed by CLAN [32].
- PLUTO, is used to make automatic parallelization based on polyhedral representation and dependencies given by CANDLY [33]. It offers compiler optimizations including an abstraction to perform high level transformations for nested affine loop structures. Its main functionality is finding affine transformations for efficient loop tiling as used in this thesis. However, it is not limited with this feature. It has the capability to make pre-vectorization, scalar privatization, and generating automatically OpenMP parallel code for multicore CPUs [33].
- The Legal Transformation Space Explorator (LetSee), is used to compute and explore affine scheduling space of programs having static control parts [34]. LetSee offers features such as:
    - An adjustable algorithm to construct legal transformation space
    - Different type of heuristic approaches for traversing legal space
    - Many other auxiliary functions such as generating transformation, graph manipulation…etc.

  In this thesis, LetSee has not been used for any loop transformation. We have chosen to use PLUTO since it offers tiling-based optimization contrary to LetSee.

- Chunky Loop Generator (CLOOG), is used to generate final source code by using transformed polyhedral representation given by PLUTO or LetSee [35].

CLAN, CANDLY and CLOOG are tools first offered by authors in [29]. However, PLUTO and LetSee are outcomes of different researches. The summary of the tool flow of POCC could be seen in figure 6.4.

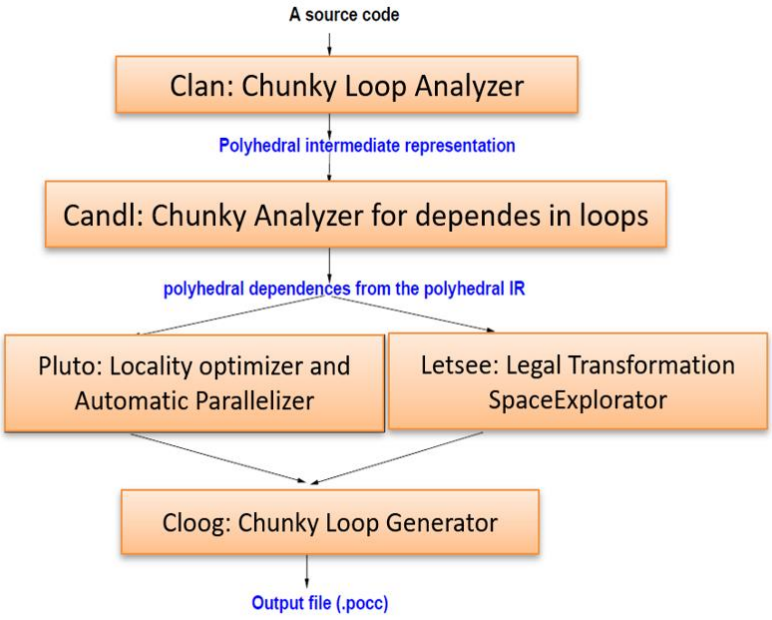


Figure 6.4. Tool Flow of POCC

The proposed system in this thesis, offers that PIPS and POCC work in sequence in order to automate source to source transformation of nested loop structures in the input source code before feeding it to the HLS tool. The following diagram illustrates all the work done before HLS. Optimized and parallelized code SCoPs in figure 6.5 is ready for being an input to the HLS tool.

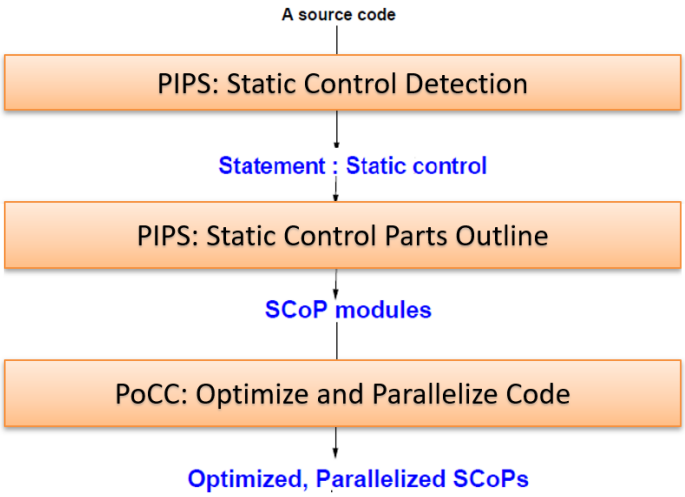


Figure 6.5. Summary of Front-End Operations of HLS

6.2.3. Accelerator Design and Verification in Vivado Tools

As stated in chapter 3.2.2, design of a hardware-based accelerator using Vivado tool has three different consecutive steps. The first step is the design of the custom accelerator IP using Vivado HLS and integrate it to FPGA design in SOC platform as the second step. Finally, communication infrastructure between CPU and FPGA has to be constructed to use the FPGA as a hardware accelerator.

In computationally intense applications, performance of the final hardware design mostly depends on the number of parallel structures, on which computation work-load is divided. In this thesis, after the front-end operations, which are SCoP extraction using PIPS and loop parallelization using POCC, it is required to copy optimized SCoP from the source code to HLS template code and arrange some minor details such as variable names by hand in order to generate the circuit that can run SCoP in a parallel manner. This is one of the reasons why the proposed system is called semi-autonomous instead of autonomous.

In this thesis work, it is required to create an HLS-friendly template code which initialize the AXI interface which provides communication for data transfer between

PL and PS side of ZYNQ-SOC device and includes some tricky adjustments in order to parallelize loop kernels because HLS can only parallelize different function calls in the main code, thus it is obligatory to rewrite the loop kernels into a function. Vivado HLS tool inherently has no capability to block loop kernels for parallel execution. This should be one of the negative sides that is necessary to be improved in Vivado HLS.

After completing the IP design of hardware accelerator, it is required to generate the bitstream file to program FPGA which is the programmable logic (PL) side of Zynq-SoC platform. This procedure is not different than the traditional FPGA design process via using Vivado Design Suite if IP designed by HLS is to be added to IP catalog of Vivado Design suite. In this thesis work, final FPGA design only consist of Hardware accelerator IP, and necessary blocks providing communication between processing system (PS) and FPGA over AXI-4 interface as shown in figure 6.6.

Last step to complete SoC system design is to configure CPU in Zynq to generate the required input data for hardware accelerator and measure designed hardware accelerator performance. In order to measure this, basic timer unit in ARM processor is used. Firstly, the timer measures the total amount of time passed from sending the first bit of data to hardware accelerator up to receiving the last bit of processed data by hardware accelerator. Then, designed system in ARM processor compares it with the reference time, which is the total amount of time if the CPU is used instead of hardware accelerator for the desired application. The basic illustration of Vivado SDK Design Flow can be seen in figure 6.7 [17].

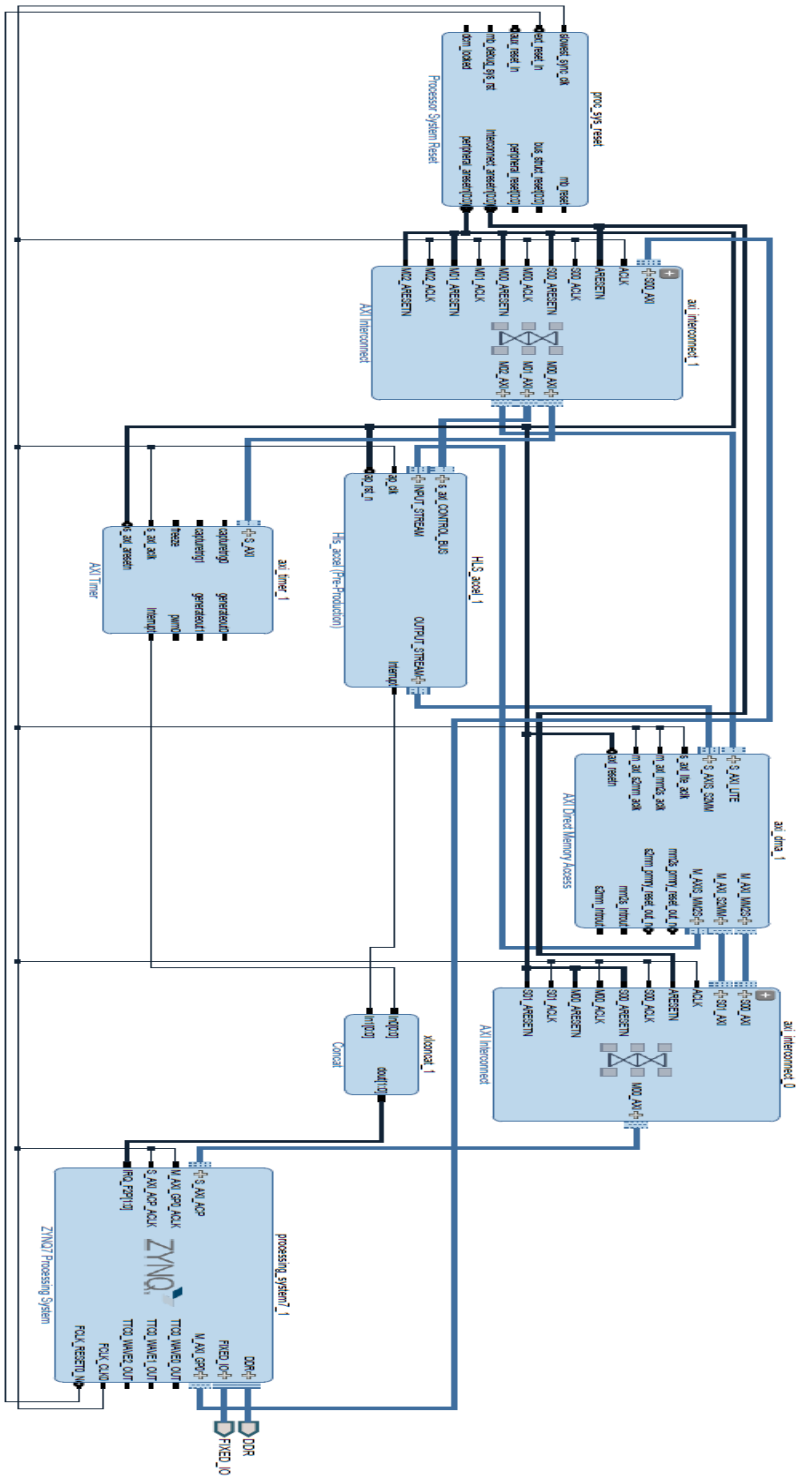


Figure 6.6. Final FPGA Block Design

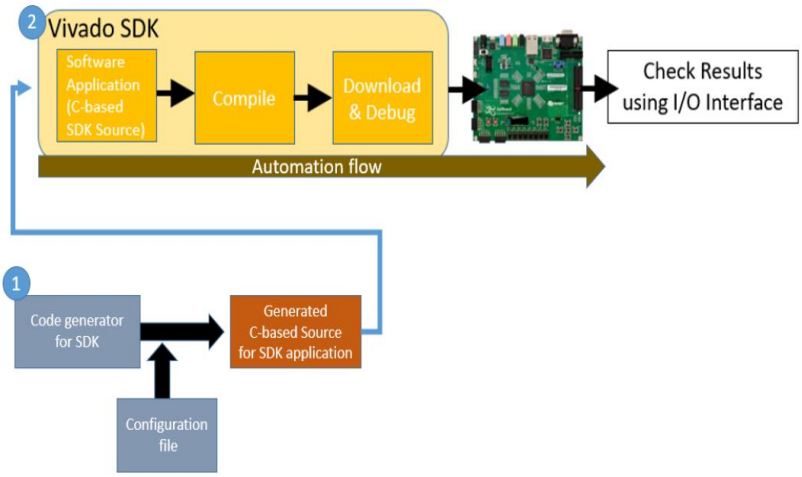


Figure 6.7. Vivado SDK Design Flow

6.3. Parallelism in Computationally Intense Applications

Exploitation of high-performance computing (HPC) platforms has paramount importance in many computationally intensive scientific computation applications since increasing computational power allows to widen the range of performance problems that could be handled. Especially, with advancements in FPGA technology, it has been widely preferred as the parallel computing platform for many applications having huge computation loads such as filtering in image processing, machine learning, etc. The biggest portions of this huge computational load are matrix multiplications and iterative stencil loops.

Hence, in this thesis work, various iterative stencil loops and matrix-matrix multiplication applications are selected in order to test the performance of the proposed semi-autonomous hardware accelerator approach.

6.3.1. Parallelism in Matrix Multiplication

Due to the being a corner stone of linear algebra and its importance in engineering applications, matrix multiplication optimization methods are repeatedly studied in the

field of both hardware and software designs. Its inherently parallelizable structure is appropriate to test and verify our proposed accelerator design. In recent years, FPGA-based matrix multiplication acceleration has become feasible alternative to software based-systems and there is plethora of research, which regard the use of matrix multiplication algorithms to analyze the performance of various FPGA-based hardware accelerator platforms [45].

All in all, matrix multiplication is one of the most suitable choice to verify and analyze our proposed semi-autonomous approach. The key idea behind the parallelization of matrix multiplication is using divide and conquer technique. Figure 6.8 shows this technique basically on matrix multiplication ($AxB = C$) by dividing first multiplier matrix horizontally and second one vertically.

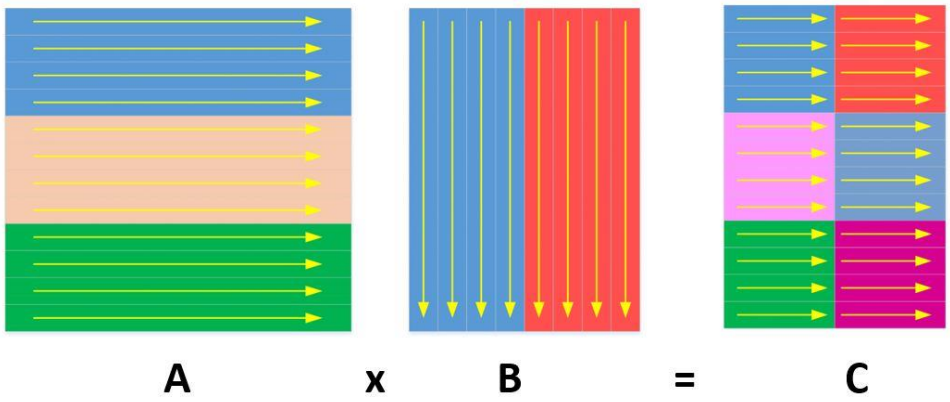


Figure 6.8. Divide and Conquer Matrix Multiplication

It is not required to do any kind of loop optimization for matrix multiplication because there is no loop carry dependency in matrix multiplication algorithm as can be easily seen in example C code in figure 6.9.

```
1  for (i = 0; i < N; i++)
2      for (j = 0; j < N; j++) {
3      sum=0;
4      for (k=0;k<N;k++)
5          sum=sum+A[i][k]+B[k][j];
6      C[i][j]=sum;
7      }
```

Figure 6.9. Square Matrix Multiplication Code

All in all, the remaining tasks before hardware designing in Vivado HLS tool are only extraction of matrix multiplication block in the given source code and prepare this code for dividing its workload among circuit blocks. Thanks to the proposed PIPS and POCC integration in this thesis, this two-step process could be done automatically without requiring a software design effort. In polyhedral model, this divide and conquer approach for loop operations is called as loop tiling.

6.3.1.1. Loop Tiling

Loop tiling is a loop transformation method, which modifies nested loop iteration bounds to iterate over tiles in other words blocks instead of completely iterating in each dimension of iteration domains. Loop tiling is also called as loop blocking or strip mining and loop interchange [46]. As can be grasped from its name, loop tiling is considered as combination of another two basic loop transformations, which are namely strip-mining and loop interchange. This transformation is used to improve data locality by fitting tile size to cache capacity or coarse-grained parallelism as used in this thesis. A tile could be of two types [47]:

- **Full Tile:** Tiling is applied on all axis of iteration domain of nested loop
- **Partial Tile:** Tiling is applied on only a subset of axis of iteration domain of nested loop

The difference between full-tiling and partial-tiling could be seen on figure 6.10 for 2D iteration domain.

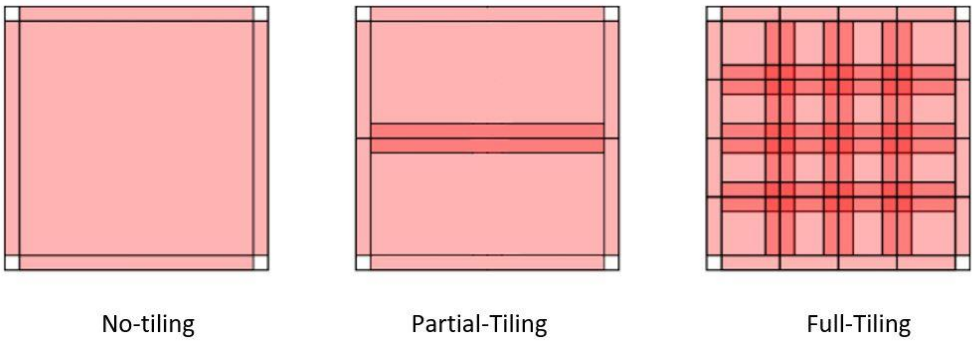


Figure 6.10. Tiling in 2D Iteration Domain

Clearly, a crucial task when doing loop tiling for course-grained parallelization is to ensure its legality. Generally, a loop transformation is legal if and only if distance vector of resultant code is legal as explained in section 4.3. POCC, which is polyhedral analysis and transformation tool used in this thesis has capability to automatically check for the legality of polyhedral transformation.

6.3.2. Parallelism in Iterative Stencil Loops

Many algorithms for scientific computations [49] and multimedia processing [48] are based on stencil computing. The computationally intensive nature of these algorithms forces designers to efficiently implement them in order to reduce total execution time. Due to their complex inner structure, which can include different kind of loop carried dependencies, hardware acceleration designs of these algorithms are traditionally considered as a difficult task. In other words, most of the times loop-tiling of these nested loops may not be directly possible.

In this thesis, a polyhedral model-based loop transformation method, which is combination of loop skewing and loop interchange is used to enable loop tiling operation for iterative stencil applications.

6.3.2.1. Iterative Stencil Loop

Iterative Stencil Loops (ISL) are a type of iterative algorithms that continuously update values on cell via correlating with its neighbors, which are cells in the same grid. The fixed pattern of neighbors is named as stencil and the function that uses values of neighbors to update a specific cell is called transition function [50]. The general pseudocode for any iterative stencil algorithms can be seen in figure 6.11.

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

Figure 6.11. Pseudocode for ISL

The number of applications using ISL is quite large as stated above, most of the scientific computation and image or video processing algorithms are based on 2D or 3D iterative stencil loops. A basic illustration of 5-point 2D iterative stencil loop can be seen in figure 6.12.

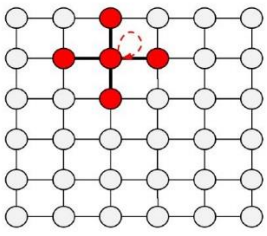


Figure 6.12. 5-Point 2D ISL

In iterative stencil loops, the contribution of neighbors is weighted by some coefficients. Obviously, these coefficients could be constant or variable. In this thesis work, both coefficient types are studied to analyze performance of offered system.

6.3.2.2. Loop Interchange

Loop interchange is exchanging the position of two loops in a nested loop structure. It is also known as loop permutation. Example codes before and after loop interchange can be seen in figure 6.13. The main usage of loop interchange is modifying the behavior of array accesses. For instance, interchanging of two loops can enable partial tiling of nested loop for parallelization as used in this thesis work.

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

Before

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

After

Figure 6.13. Loop Interchange Example

Obviously, loop interchange is legal if and only if distance vectors remain lexicographically positive after the interchange. Nested loop in figure 6.14 cannot be interchanged because its distance vector is $(1, -1)$ and after loop interchange it becomes $(-1, 1)$ which is not lexicographically positive [46].

```
do i = 2, n
 do j = 1, n-1
  a[i,j] = a[i-1,j+1]
 end do
end do
```

Figure 6.14. Non-Interchangeable Loop Example

6.3.2.3. Loop Skewing

Loop skewing is a polyhedral transformation which changes the lower and upper bounds of inner loop with respect to the outer loop in a nested loop. Skewing iteration space of nested loop structure is a natural result of changing of loop bounds. This transformation is useful in order to expose convenience to parallelism for nested loops which inner loop has a dependency on outer loop. This transformation and its results could be understood easily examining an example code in figure 6.15 [46].

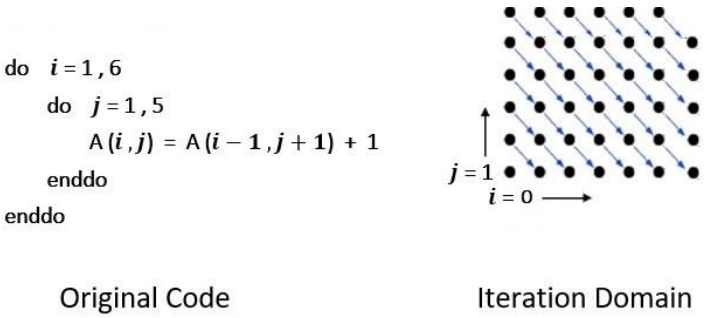


Figure 6.15. Loop Skewing Example Code

Blue arrows in figure 6.15 represent distance vector, which is $(1, -1)$, of the original code. Distance vector and iteration domain in figure 6.15 show that neither loop can be parallelized to accelerate nested loop algorithm. It should be note that formal definition of loop carried dependency cannot be used to decide whether a loop could be parallelized or not since the number of distance vector in this example is only one.

Loop skewing is performed by adding loop index of outer loop to inner loop bounds and subtracting same skew value from index of statements in inner loop body. This subtraction preserves correctness of the program. The effect of skewing on inner loop is to deviate loop bounds relative to outer loop bounds and increase distance to outer loop in the same manner. In other words, assume a distance vector (x, y), after skewing it changes to$(x, x + y)$. Loop skewing is always a legal transformation because it protects lexicographic order of the previous distance vectors. Application

of skewing original code in figure 6.15 can be seen on figure 6.16. It is worth noting that, loop index variables are changed from (i, j) to (i', j') after loop skewing in figure 6.16.

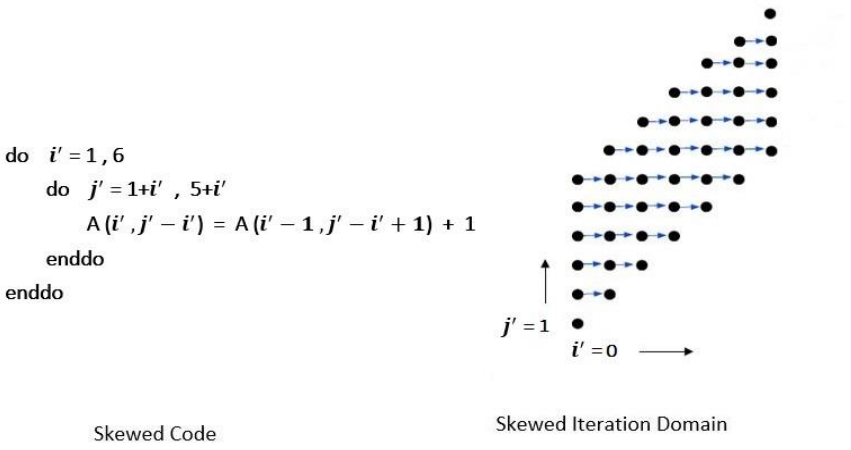


Figure 6.16. Skewed Code and Iteration Domain

After loop skewing, new distance vector is $(1,0)$. As can be seen on skewed iteration domain in figure 6.16, inner loop cannot be parallelized. On the other hand, at this point interchanging iteration domain tracing order, thanks to the loop interchanging, can enable inner loop parallelization as shown in figure 6.17.

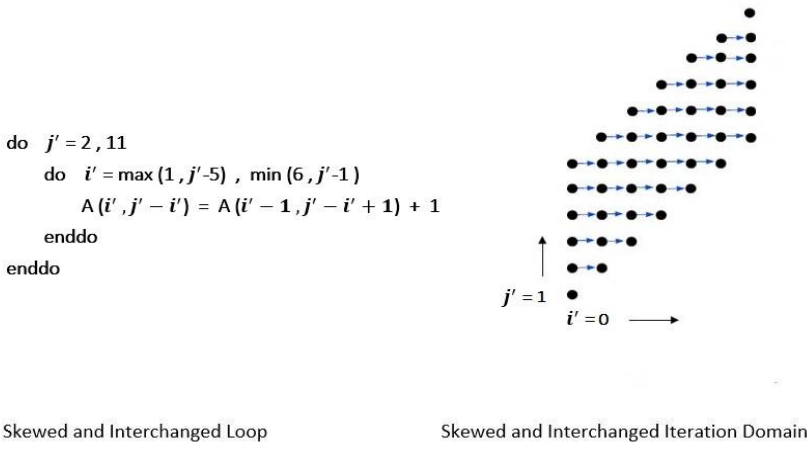


Figure 6.17. Skewed-Interchanged Code and Iteration Domain

It is clear that, after loop interchange, adjusting loop bounds is not straightforward. Each loop has to take care of upper and lower bounds of the other loops.

6.3.2.4. Combining Loop Skewing with Tiling

Skewing a loop can enable loop tiling that was not possible in the original loop because of its dependence structure. Skewing a loop and applying a tiling is called as skewed tiling. The order of these transformation is crucial. The reverse order which is tiling a loop and then skewing generate a tiled loop whose bounds are only skewed instead of all iterations in original nested loop. Although resultant code seems to be skewed relative to original code, this transformation is redundant because original iteration domain dimension does not change after loop skewing.

Applying loop skewing and tiling in correct order means tiling transformation operates on skewed iteration domain. In this case, tiles may not have the same rectangular shape due to the skewing of iteration space as shown on example iteration space in figure 6.18.

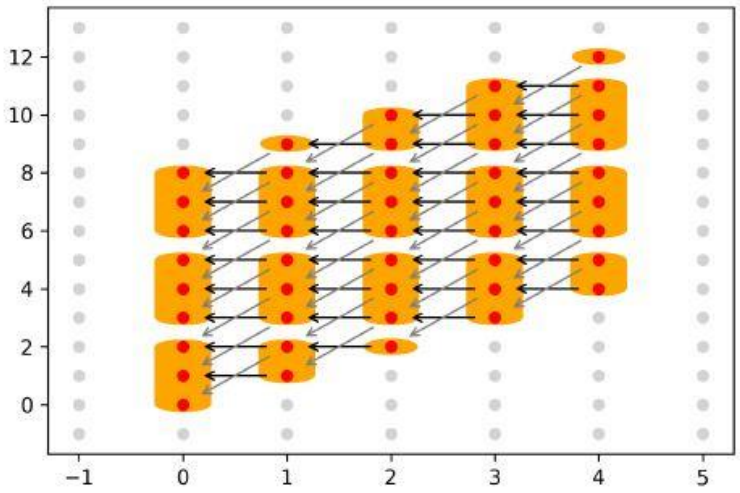


Figure 6.18. Skewed Tiling Iteration Space

Polyhedral Compiler Collection (PoCC) tool that is a polyhedral analysis and loop transformation tool used in this thesis work unfortunately has no capability to apply loop skewing on given nested loops although it can automatically tile these. In this thesis, loop skewing is applied manually. This is another reason why our proposed approach is called as semi-autonomous. The other reason is the manual generation of HLS template code in order to parallelize given tiled nested-loops as mentioned in section 6.2.3.

CHAPTER 7

PERFORMANCE EVALUATION

7.1. Test Environment

In this thesis work, the proposed semi-autonomous approach is designed, implemented and verified on Xilinx ZC-702 evaluation kit. ZC-702 is a complete development kit for developers working on designing and testing different kind of systems. The ZC-702 board include Xilinx Zynq-7000 All programmable SoC and all required interfaces supporting broad range of applications. In this section, basic background information about ZC-702 evaluation kit, and Zynq SoC platform are given.

7.1.1. Xilinx ZC-702 Evaluation Board

ZC-702 is a multi-purpose development board for both software and hardware developers to design and verify various applications. This board support many target applications such as video processing, hardware/software accelerators, Linux/Android development, embedded arm processor design, etc.

Basic features of ZC-702 could be listed as follows [63]:

- Zynq-7000 All Programmable SoC device (XC7Z020-CLG484)
- 1 GB DDR3 Memory
- 128 Mb QSPI Flash Memory
- USB OTG 2.0
- 10/100/1000 Ethernet interface with RJ-45 connector
- Onboard JTAG programmer
- HDMI codec
- Fixed 200 MHz LVDS oscillator
- I2C programmable LVDS oscillator

Figure 7.1 illustrates ZC-702 Evaluation board with all interfaces and features [63].

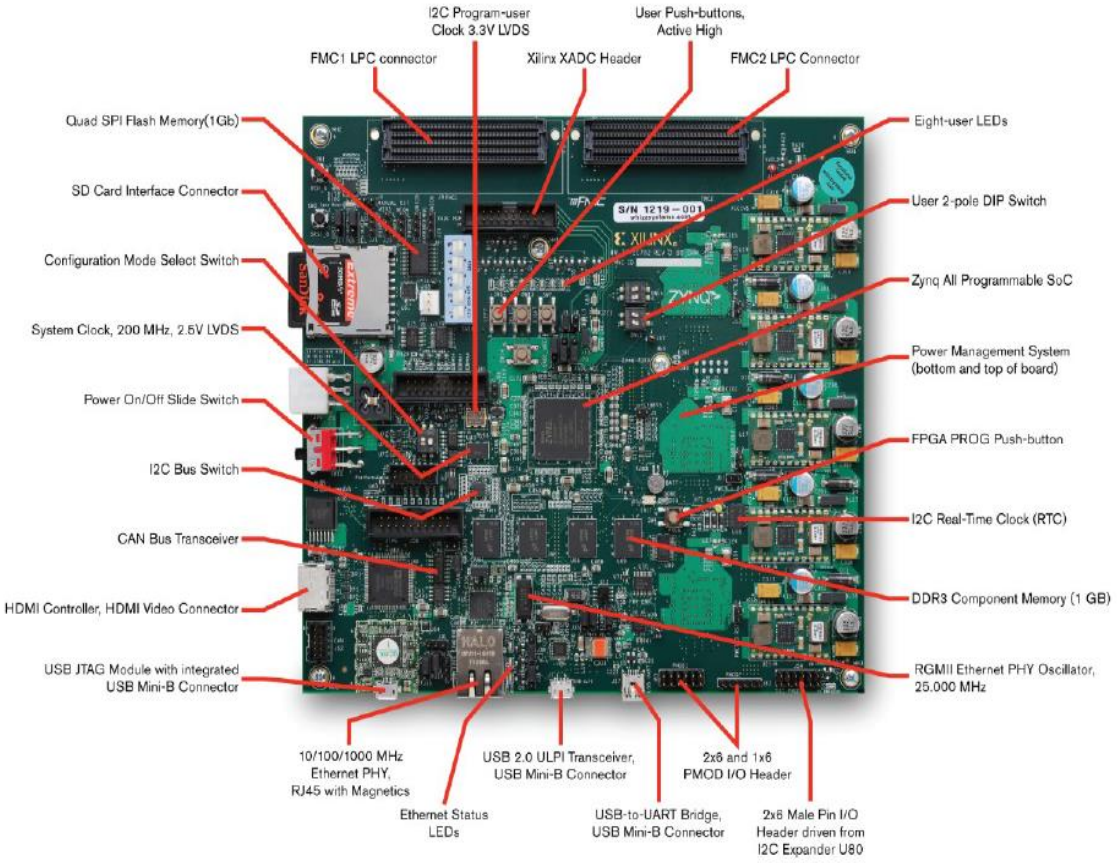


Figure 7.1. ZC-702 Evaluation Board

7.1.2. Zynq-7000 SoC

Xilinx Zynq- XC7Z20 integrated circuit (IC) is composed of both dual ARM Cortex-AP CPU called as processing system (PS) and Artix-7 base FPGA called as programmable logic (PL). PS and PL communicate over AXI interface. Figure 7.2 illustrates the block diagram of ZYNQ-7000 series IC [10].

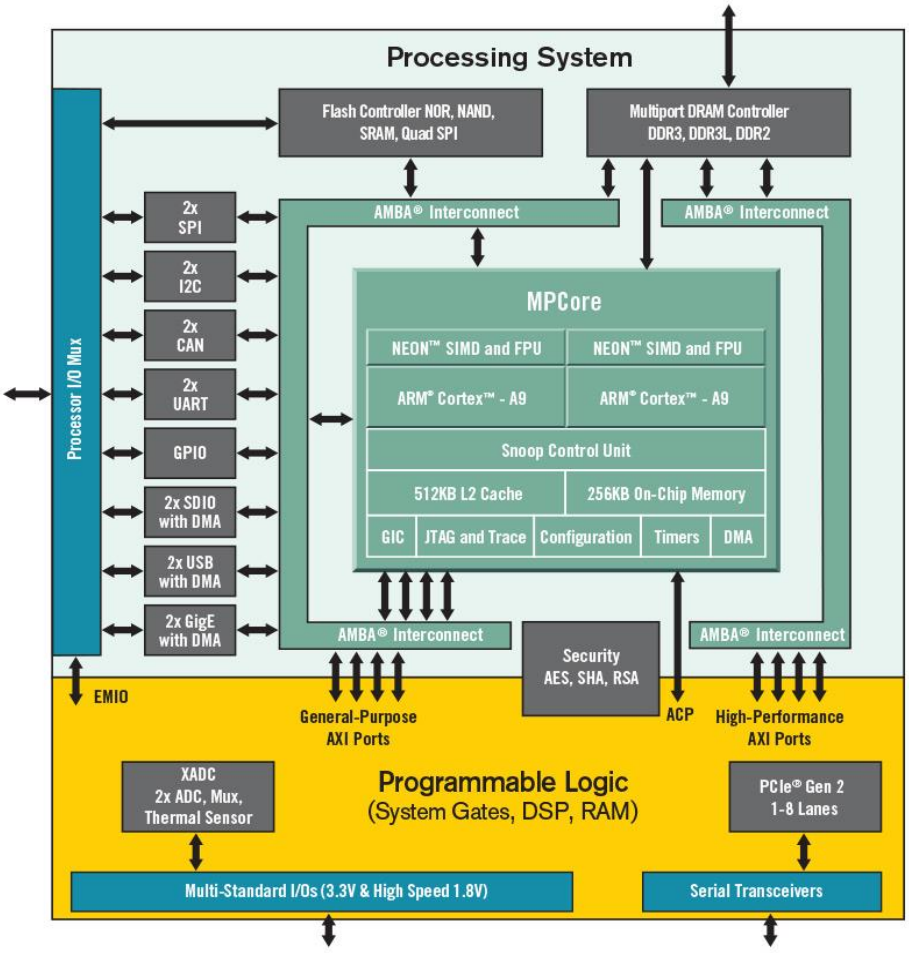


Figure 7.2. Block Diagram of Zynq-7000

7.1.2.1. Processing System (PS)

The application processor unit (APU) has a dual ARM Cortex-A9 MPCore with version 7 ARM ISA that works on maximum 1GHz frequency. It has 1GB DDR3 memory and has a 128 Mb QSPI Flash. There is a 32KB L1 4-way set associative instruction and data cache, and a 512KB L2 8-way set associative data cache which supports byte-parity [10].

7.1.2.2. Programmable Logic (PL)

PL is based on the artrix-7 FPGA logic of Xilinx. The PL contains configurable logic block (CLB) which has a 6-input look-up table (LUT), memory capability within the LUT, Register/shift register and adders. It consists 36KB block RAM, and DSP with 48 bit high-resolution. The PL resources consist of 13,300 Logic slice, 53,200 LUTs, 140 BRAM and 220 DSPs [10].

7.1.2.3. PS-PL Communication

Zynq-7000 series devices use advanced extensible Interface (AXI) to set up communication infrastructure between PL and PS. There are three types of AXI. First is AXI4, which is for high-performance memory-mapped interfaces. Secondly, AXI4-Lite is the low-throughput memory mapped interface. Lastly, AXI4-Stream is used for high-speed streaming data for video/audio processing [64]. The Figure 7.3 is an AXI interface diagram showing a streaming data transfer between the PS and PL. In general, the AXI4-Stream interface is used together with a Direct Memory Access (DMA) controller to transfer large amount of data from the processor (ARM) to the programmable logic (FPGA). This data is transformed as vector data on the software side. The DMA controller reads the vector data from the memory and streams it to FPGA through the AXI4-Stream interface.

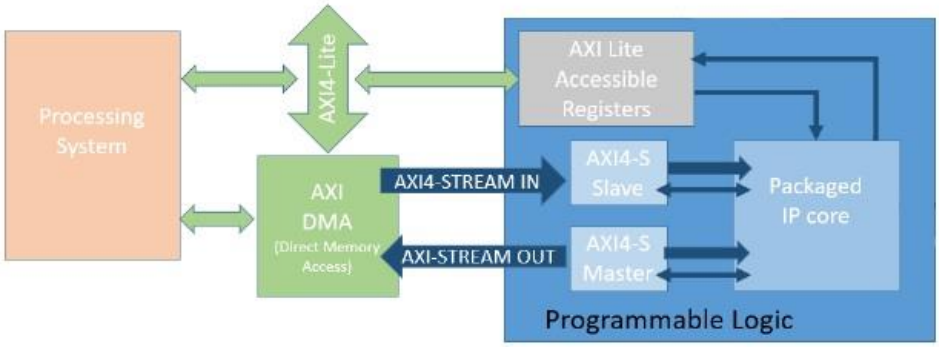


Figure 7.3. AXI Interface on Zynq SoC

7.2. Overview of Test Scenarios

As mentioned in chapter 6, matrix multiplication and iterative stencil loops are favorable choices in order to test the performance of any accelerator design for computationally intense applications. In this thesis work, they have been comprehensively employed in different test scenarios.

For each loop kernel having different size and parallelization degree are used in our tests, in five different scenarios:

Simple CPU Execution:

In order to examine the performance of the hardware accelerator, sequential execution time of the given algorithm on CPU, which is ARM Cortex A9, has to be known. Also, it should be noted that given algorithm is compiled by Xilinx SDK compiler with default optimization level setting which is optimize more (-O2) level. Abbreviation used for this scenario in section 7.3 and 7.4 is "CPU Exe.".

Simple HLS Design:

In this scenario, after SCoP extraction using PIPS, any polyhedral or inherent HLS optimizations is applied on the loop kernel. FPGA IP for each loop kernel is generated using HLS flow without any optimization. Abbreviation used for this scenario in section 7.3 and 7.4 is "HLS w/o Opt.".

HLS Design with Inherent Vivado Optimizations:

In this scenario, only inherent HLS instruction-based optimizations which are explained in section 3.3 are used. In other words, although given loop kernel is optimized in this scenario, loop parallelization degree is still one. In order to take advantage of these optimizations, it is required to insert pragma "pipeline" in the loop body. Usage of pipeline pragmas of Vivado HLS tool can be seen

in figure 7.4. Abbreviation used for this scenario in section 7.3 and 7.4 is “HLS w/ vivOpt.”

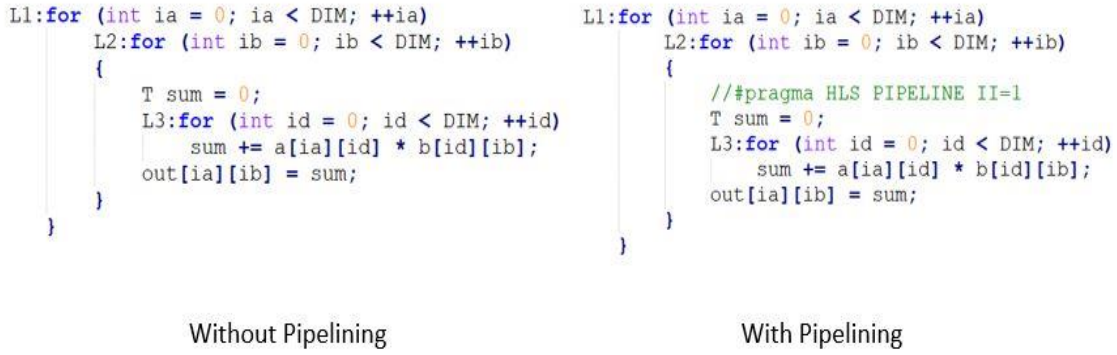


Figure 7.4. Vivado Pipeline Pragma Usage

HLS Design with Polyhedral Optimizations:

In this scenario, previously mentioned polyhedral model-based optimizations which are loop tiling and skewing are used instead of inherent HLS optimization to compare the performance of the proposed system with results of previous scenarios. Abbreviation used for this scenario in section 7.3 and 7.4 is “HLS w/ poly.”

HLS Design with its Inherent Vivado and Polyhedral Optimizations:

In this scenario, previous two optimizations are combined. Abbreviation used for this scenario in section 7.3 and 7.4 is “HLS w/ viv_poly.”.

It should be noted that operational clock frequency for both CPU and FPGA is set to 100MHz and all execution time results are in unit of clock cycle for all tests. Also, in order to download CPU bit stream and hardware description file to Zynq SoC, on-board USB-JTAG module of ZC702 has been used. Besides, USB-to-UART bridge of ZC702 which is a serial port between Zynq PS side and external world is employed

to read experimental results from ARM processor. Thus, there are two different connections between computer that executes Vivado and evaluation board as shown in figure 7.5.

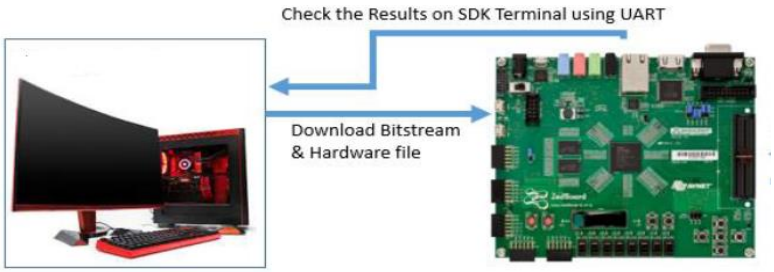


Figure 7.5. Connections between Eval. Board and Comp.

In this thesis, a code block on ARM processor has been designed to measure the execution time of a given loop kernel for both ARM and hardware accelerator executions and compare these results in terms of speed and correctness. Then, this code block sends the results to external world over UART port with 115200 baud-rate. Figure 7.6 demonstrates the results of an example execution of an hardware accelerator on the serial port of Vivado SDK.

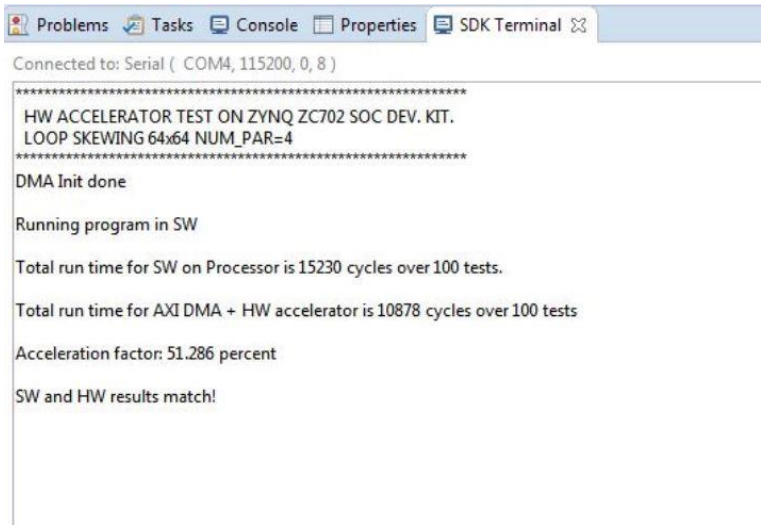
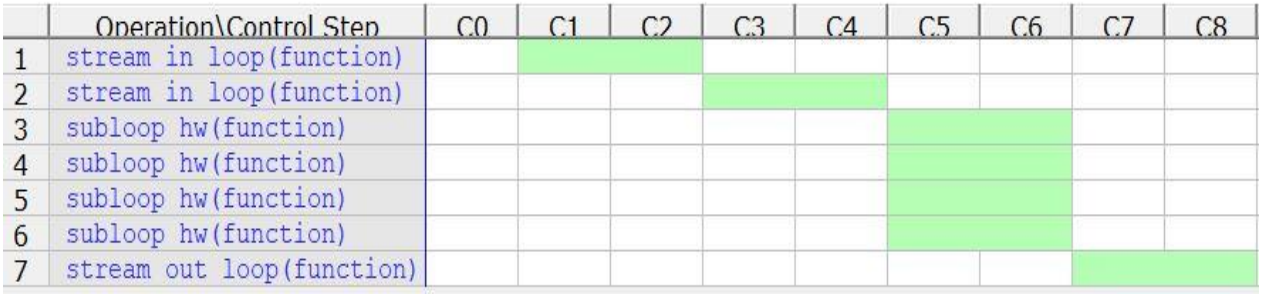


Figure 7.6. Results on UART port of SDK

In order to ensure whether compiled HLS code suits the desired parallelization degree or not, it is required to check operation\control step diagram provided by Vivado HLS tool. If the example diagram in figure 7.7 is examined, parallelization degree of loop kernel is four as can be easily understand by the C5 and C6 control steps. “stream_in_loop” operation represents reading input data from AXI interface and “stream_out_loop” operation represents writing output data to AXI interface.



	Operation\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8
1	stream in loop(function)									
2	stream in loop(function)									
3	subloop hw(function)									
4	subloop hw(function)									
5	subloop hw(function)									
6	subloop hw(function)									
7	stream out loop(function)									

Figure 7.7. Operation\Control Step Diagram

7.3. Evaluation Using Matrix-Matrix Multiplication

Square matrix-matrix multiplications of different sizes are tested applying different parallelization degrees in order to examine the performance of proposed architecture. After running PIPS-POCC integrated tool with usual matrix-matrix multiplication code, matrix multiplication code has been tiled to enable course-grained loop parallelization as shown in figure 7.8. POCC tool also changes loop index variable names and re-declare these variables. If the output code of PIPS-POCC is examined, parallelization degree of loop kernel can be adjusted via changing b1 and b2 variables.

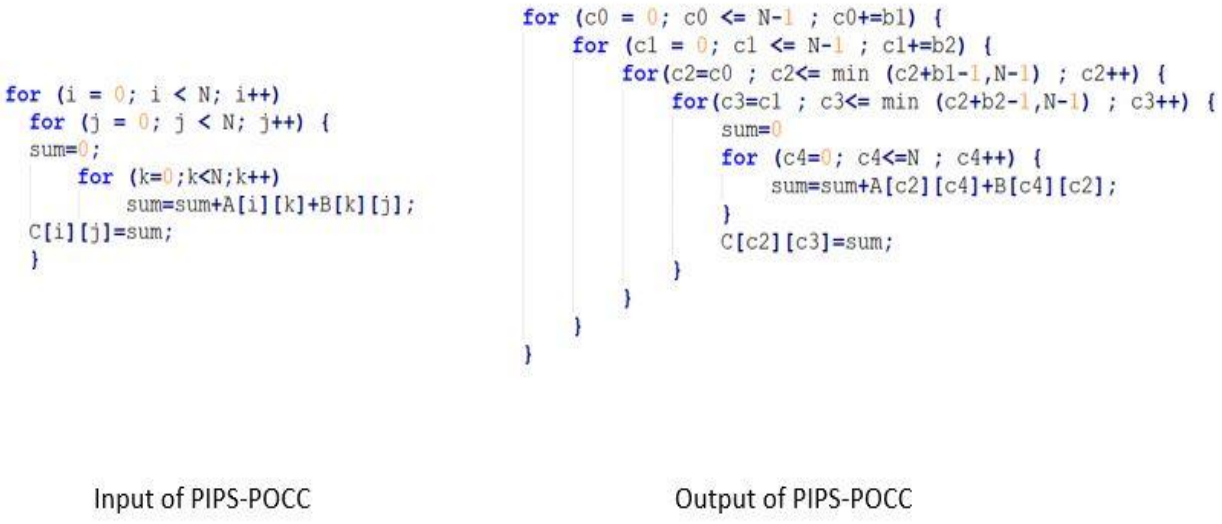


Figure 7.8. Tiled Code of Matrix Multiplication

7.3.1. Performance and Resource Usage Analysis

The hardware accelerator has been tested by using floating-point matrix multiplication algorithm with three different sizes, which are 64x64, 128x128, 256x256 and experimental and simulation results are analyzed for three different loop parallelization degree which are 1, 2 and 4.

Table 7.1 and table 7.2 (table 7.3 and table 7.4) show execution time measurement results and FPGA resource usages for 64x64 (128x128) matrix multiplication respectively.

Table 7.1. *Exe. Time Measurement Results of 64x64 Matrix Multip*

Parallelization Degree	**CPU Exe.**	**HLS w/o Opt.**	**HLS w/ vivOpt.**	**HLS w/ poly.**	**HLS w/ viv_poly.**
1	184850	1814544	79856	-	-
2	-	-	-	690832	37351
4	-	-	-	171184	19799

Unit: Clock Cycle

Table 7.2. *FPGA Resource Usage of 64x64 Matrix Multip*

Parallelization Degree	**BRAM (%)**		**DSP (%)**		**FF (%)**		**LUT (%)**	
1 (viv_opt)	40		4		12		20	
	poly	**viv_poly**	**poly**	**viv_poly**	**poly**	**viv_poly**	**poly**	**viv_poly**
2	39	40	4	9	1	17	5	19
4	39	40	9	18	2	42	9	40

Table 7.3. *Exe. Time Measurement Results of 128x128 Matrix Multip.*

Parallelization Degree	**CPU Exe.**	**HLS w/o Opt.**	**HLS w/ vivOpt.**	**HLS w/ poly.**	**HLS w/ viv_poly.**
1	920367	7326480	482007	-	-
2	-	-	-	2550321	203767
4	-	-	-	1404240	111255

Unit: Clock Cycle

Table 7.4. *FPGA Resource Usage of 128x128 Matrix Multip.*

Parallelization Degree	**BRAM (%)**		**DSP (%)**		**FF (%)**		**LUT (%)**	
1 (viv_opt)	68		4		12		20	
	poly	**viv_poly**	**poly**	**viv_poly**	**poly**	**viv_poly**	**poly**	**viv_poly**
2	61	68	4	9	1	24	6	34
4	60	68	9	18	2	48	9	69

When matrix size has been increased to 256x256, following results in table 7.5 and 7.6 are obtained from Vivado HLS simulations. Because of limited BRAM and LUT capacities of zynq programmable logic, it is not possible to verify the proposed accelerator system with 256x256 floating point matrix multiplication. However, these simulation results help to grasp how to FPGA resource use change in response to

changing matrix size and verify trends of execution time with changing parameters on a larger data set.

Table 7.5. *Exe. Time Simulation Results of 256x256 Matrix Multip.*

Parallelization Degree	**CPU Exe.**	**HLS w/o Opt.**	**HLS w/ vivOpt.**	**HLS w/ poly.**	**HLS w/ viv_poly.**
1	13961349	83437584	8520983	-	-
2	-	-	-	45648400	3462295
4	-	-	-	11381392	1805815

Unit: Clock Cycle

Table 7.6. *FPGA Resource Usage of 256x256 Matrix Multip. (Simulation Result)*

Parallelization Degree	**BRAM (%)**		**DSP (%)**		**FF (%)**		**LUT (%)**	
1 (viv_opt)	274		4		23		36	
	poly	**viv_poly**	**poly**	**viv_poly**	**poly**	**viv_poly**	**poly**	**viv_poly**
2	274	274	4	9	1	45	6	64
4	274	274	9	18	2	90	9	124

Results for all 256x256, 128x128, 64x64 floating point matrix multiplication shows that if a hardware accelerator is designed for matrix multiplications using HLS tool without any optimization (HLS w/o Opt.), designed accelerator is dramatically slower than the usual CPU sequential executions as can be seen in table 7.7. If the size of matrices increases, it is examined that deceleration ratio decreases. However, these are certainly undesirable results.

Table 7.7. *Comparisons of CPU Exe. And HLS w/o Opt.*

Matrix Size	**CPU Exe.**	**HLS w/o Opt.**	***Deceleration Ratio (%)***
64x64	184850	1814544	881
128x128	920367	7326480	696
256x256	13961349	83437584	498

Unit: Clock Cycle

On the other hand, when HLS inherent optimization methods are applied on loop bodies via easily adding pipelining pragma, shown in figure 7.4. HLS based accelerator designs for all matrix sizes reach to acceptable performance without any course-grained parallelism on nested loop structure due to the instruction level pipelining on loop bodies as shown in table 7.8. Also, it should be taken into consideration that, when the matrix sizes increase, acceleration ratio decrease due to the increase in the communication overhead between ARM processor and FPGA while transferring input matrices and resultant matrix after multiplication and memory operations overhead.

Table 7.8. *Comparisons of CPU Exe. and HLS w/ vivOPT.*

Matrix Size	**CPU Exe.**	**HLS w/ vivOpt.**	***Acceleration Ratio (%)***
64x64	184850	79856	132
128x128	920367	482007	90
256x256	13961349	8520983	63

Unit: Clock Cycle

When the proposed front-end polyhedral model-based optimization and inherent HLS pipelining are combined, resultant matrix multiplication IP block gains both course

grained parallelism and instruction level pipelining at loop body. Hence, designed hardware accelerator is more effective in terms of execution time. Table 7.9 demonstrates this effectiveness via comparing best results of offered accelerator with highest degree of parallelism, that is four in this work, and HLS optimized results. Theoretically, if the number of parallel processing blocks is four, system should be four times faster than the sequential counterpart. Obviously, results deviate from theoretical values in practice due to the unavoidable communications and some minor operations (port initializations, timer setup, etc.) overheads as can be seen in table 7.9. Also, it can be said that if matrix size increases communication overhead between CPU and FPGA becomes less comparable with computational loads. In other words, performance offered hardware accelerator is higher when input matrix is larger as could be understand acceleration ratios in table 7.9.

Table 7.9. *Comparisons of HLS w/vivOpt. and HLS w/viv_poly*

Matrix Size	**HLS w/ vivOpt.**	**HLS w/ viv_poly.**	***Acceleration Ratio (%)***
64x64	79856	19799	303
128x128	482007	111255	334
256x256	8520983	1805815	371

Unit: Clock Cycle

The major drawback of our architecture is the increase in FPGA resource usage. If table 7.2, table 7.4 and table 7.6, are examined when degree of parallelism increases, FPGA resource usage also increases with the same ratio except BRAM. Because usage of BRAM mostly depends on matrix size.

7.4. Evaluation Using Iterative Stencil Loops

There are many algorithms using iterative stencil loops. In order to test and verify a hardware accelerator with iterative stencil loops, we have gotten a kernel from

Polybench benchmark suite [25]. In this thesis, Jacobi 2-D stencil kernel, which comes from Polybench suite is selected to test our system's performance.

7.4.1. Jacobi 2D Stencil with Constant Coefficients

Constant Coefficient Jacobi 2D stencil is an iterative computational algorithm which calculate mean value of five points accessed in a specific pattern shown in figure 7.9. Jacobi 2D is the kernel that frequently used in many linear algebra and image processing algorithms.

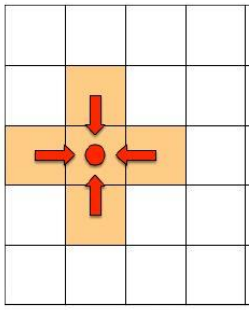


Figure 7.9. Jacobi 2D Computational Kernel

Let's briefly analyze the Jacobi 2D stencil code for square matrix kernel used to test hardware accelerator in this thesis in figure 7.10. Its legal distance vectors are (1,0) and (0,1). Hence, neither loops in Jacobi 2D stencil cannot be parallelized without any modification, in other words, full or partial loop tiling is not possible to enable course grained parallelism because each loop have loop carry dependency.

```
1  L1:for (int k = 1 ; k < DIM-1; k++)
2      L2:for (int l =1 ; l < DIM-1; l++)
3          {
4          a[k][l] = (a[k-1][l] + a[k+1][l] + a[k][l-1] + a[k][l+1])/4;
5          }
```

Figure 7.10. Jacobi 2D Stencil Code

When loop skewing and loop interchange which are reviewed in section 6.3.2 is applied on Jacobi 2D stencil by hand before feeding it to offered PIPS-POCC integrated tool, distance vector of transformed Jacobi 2D stencil, that can be seen in figure 7.11, becomes (1,0) and (1,1). Now, the inner loop could be parallelized because it has now no loop carried dependency.

```
L1:for (int k = 1 ; k<= DIM-1; k++)
    L2:for (int l=1+k ; l<=DIM-1+k; l++)
        {
        a[k][l-k] = (a[k-1][l-k] + a[k+1][l-k] +
        a[k][l-1-k] + a[k][l+1-k])/4;
        }
```

Skewed Jacobi 2D

```
L1:for (int l = 2 ; l <= 2*DIM-2; l++)
    L2:for (int k=max(1,l-DIM+1) ; l<= min(DIM-1,l-1); k++)
        {
        a[k][l-k] = (a[k-1][l-k] + a[k+1][l-k] +
        a[k][l-1-k] + a[k][l+1-k])/4;
        }
```

Skewed and Interchanged Jacobi 2D

Figure 7.11. Transformed Jacobi 2D Stencil Code

After these transformations, Jacobi 2D stencil is ready for partial loop tiling to enable course grained parallelism. Partial tiled code, which is generated automatically by POCC, could be seen on figure 7.12. Parallelization degree is determined by changing variable “b1” in figure 7.12.

```
for ( int ll=2 ; ll < 2*DIM-2 ; ll+=b1)
    for (int l = ll ; l<= min(ll+b1-1,2*DIM-2) ; l++)
        for (int k=max(1,l-DIM+1) ; l<= min(DIM-1,l-1); k++)
            {
            a[k][l-k] = (a[k-1][l-k] + a[k+1][l-k] +
            a[k][l-1-k] + a[k][l+1-k])/4;
            }
```

Figure 7.12. Tiled Jacobi 2D Stencil

7.4.1.1. Performance and Resource Usage Analysis

Offered hardware accelerator has been tested via using constant coefficient transformed Jacobi 2D iterative stencil loop kernel with 3 different dimensions which are 64x64, 128x128, 256x256 and experimental and simulation results are analyzed for 3 different loop parallelization degree which are 1,2,4.

Table 7.10 and table 7.11 (table 7.12 and table 7.13) show execution time measurement results and FPGA resource usages for constant coefficient transformed Jacobi 2D having size 64x64 (128x128) respectively.

Table 7.10. *Exe. Time Measurement Results of 64x64 Cons. Coeff. Jacobi 2D*

Parallelization Degree	**CPU Exe.**	**HLS w/o Opt.**	**HLS w/ vivOpt.**	**HLS w/ viv_poly.**
1	99654	117892	78715	-
2	-	-	-	45258
4	-	-	-	28250

Unit: Clock Cycle

Table 7.11. *FPGA Resource Usage of 64x64 Cons. Coeff. Jacobi 2D*

Parallelization Degree	**BRAM (%)**	**DSP (%)**	**FF (%)**	**LUT (%)**
1 (viv_opt)	6	2	1	4
2	6	4	2	7
4	6	9	3	11

Table 7.12. *Exe. Time Measurement Results of 128x128 Cons. Coeff. Jacobi 2D*

Parallelization Degree	**CPU Exe.**	**HLS w/o Opt.**	**HLS w/ vivOpt.**	**HLS w/ viv_poly.**
1	398676	497478	321210	-
2	-	-	-	184744
4	-	-	-	115880

Unit: Clock Cycle

Table 7.13. *FPGA Resource Usage of 128x128 Cons. Coeff. Jacobi 2D*

Parallelization Degree	BRAM (%)	DSP (%)	FF (%)	LUT (%)
1 (viv_opt)	25	2	1	4
2	25	4	2	7
4	25	9	3	11

When matrix size has been increased to 256x256, following results in table 7.14 and 7.15 are gotten experimentally instead of Vivado HLS contrary to matrix multiplication example in section 7.2. Since, number of input matrices is only one for Jacobi 2D stencil benchmark however this number is two for matrix multiplication. This difference between Jacobi 2D kernel and matrix multiplication results in reduction of BRAM usage.

Table 7.14. *Exe. Time Measurement Results of 256x256 Cons. Coeff. Jacobi 2D*

Parallelization Degree	CPU Exe.	HLS w/o Opt.	HLS w/ vivOpt.	HLS w/ viv_poly.
1	1586072	2060593	1297722	-
2	-	-	-	746280
4	-	-	-	459288

Unit: Clock Cycle

Table 7.15. *FPGA Resource Usage of 256x256 Cons. Coeff. Jacobi 2D*

Parallelization Degree	BRAM (%)	DSP (%)	FF (%)	LUT (%)
1 (viv_opt)	92	2	1	5
2	92	4	2	7
4	92	9	3	12

Results for all 256x256, 128x128, 64x64 constant coefficient Jacobi 2D stencil kernel shows that hardware accelerator designed by only HLS inherent optimization have acceptable performance without polyhedral optimizations as shown in Table 7.16.

Table 7.16. *Comparisons of CPU Exe and HLS w/ vivOPT*

Matrix Size	**CPU Exe.**	**HLS w/ vivOpt.**	***Acceleration Ratio (%)***
64x64	99654	78715	26
128x128	398676	321210	24
256x256	1586072	1297722	22

Unit: Clock Cycle

If course grained parallelism is enabled by offered polyhedral optimization methods, performance of hardware accelerator for Jacobi 2D stencil kernel becomes higher in terms of execution time as shown in table 7.17 which comperes results of hardware accelerator only optimized by HLS tool and optimized by offered system with 4 parallelism degree for various matrix sizes. However, it should not be forgotten that FPGA resource usage of hardware accelerator having higher degree of parallelism is also higher.

Table 7.17. *Comparisons of HLS w/ and HLS w/viv_poly*

Matrix Size	**HLS w/ vivOpt.**	**HLS w/ viv_poly.**	***Acceleration Ratio (%)***
64x64	78715	28250	173
128x128	321210	115880	177
256x256	1297722	459288	183

Unit: Clock Cycle

7.4.2. Jacobi 2D Stencil with Variable Coefficients

As stated in section 6.3.2.1, in stencil computations, contribution of the neighbor could be weighed by variable coefficients instead of constant. In this case, stencil weights

could variate between time-steps or from one grid to another one during execution. These weights are stored in separate memory location during computations that definitely causes an extra memory traffic overhead.

In this part of thesis, Jacobi 2D stencil benchmark is modified as shown in figure 7.13 in order to test performance of offered hardware accelerator via using a coefficient matrix.

```
1  L1:for (int k = 1 ; k<= DIM-1; k++)
2      L2:for (int l=1 ; l<=DIM-1; l++)
3          {
4          a[k][l] = b[k-1][l]*a[k-1][l] + b[k+1][l]*a[k+1][l] +
5                    b[k][l-1]*a[k][l-1] + b[k][l+1]a[k][l+1];
6          }
```

Figure 7.13. Jacobi 2D Stencil with Var. Coeff.

The optimization procedure of variable coefficient Jacobi 2D stencil is same with constant coefficient one studied in previous section. Step by step transformation of variable coefficient Jacobi 2D stencil loop kernel could be seen in Figure 7.14.

7.4.2.1. Performance and Resource Usage Analysis

Offered hardware accelerator has been tested via using variable coefficient transformed Jacobi 2D iterative stencil loop kernel with 3 different which are 64x64, 128x128, 256x256 and experimental and simulation results are analyzed for 3 different loop parallelization degree which are 1,2,4.

```
L1:for (int k = 1 ; k<= DIM-1; k++)
    L2:for (int l=1+k ; l<=DIM-1+k; l++)
        {
         a[k][l-k] = b[k-1][l-k]*a[k-1][l-k] + b[k+1][l-k]*a[k+1][l-k] +
                     b[k][l-1-k]*a[k][l-1] + b[k][l+1-k]a[k][l+1];
        }
```

Skewed Var. Coeff. Jacobi 2D

```
L1:for (int l = 2 ; l <= 2*DIM-2; l++)
    L2:for (int k=max(1,l-DIM+1) ; l<= min(DIM-1,l-1); k++)
        {
        a[k][l-k] = b[k-1][l-k]*a[k-1][l-k] + b[k+1][l-k]*a[k+1][l-k] +
                    b[k][l-1-k]*a[k][l-1] + b[k][l+1-k]a[k][l+1];
        }
```

Skewed and Interchanged Var. Coeff. Jacobi 2D

```
for ( int ll=2 ; ll < 2*DIM-2 ; ll+=bl)
    for (int l = ll ; l<= min(ll+bl-1,2*DIM-2) ; l++)
        for (int k=max(1,l-DIM+1) ; l<= min(DIM-1,l-1); k++)
            {
            a[k][l-k] = b[k-1][l-k]*a[k-1][l-k] + b[k+1][l-k]*a[k+1][l-k]
                        b[k][l-1-k]*a[k][l-1] + b[k][l+1-k]a[k][l+1];
            }
```

Tiling Enabled Var. Coeff. Jacobi 2D

Figure 7.14. Loop transformation of Jacobi 2D Stencil with Var. Coeff.

Table from 7.18 to 7.23 demonstrate execution time measurement results and FPGA resource usages for constant coefficient transformed Jacobi 2D having size 64x64, 128x128, 256x256 respectively.

Table 7.18. *Exe. Time Measurement Results of 64x64 Var. Coeff. Jacobi 2D*

Parallelization Degree	**CPU Exe.**	**HLS w/o Opt.**	**HLS w/ vivOpt.**	**HLS w/ viv_poly.**
1	119568	142744	93614	-
2	-	-	-	54326
4	-	-	-	33611

Unit: Clock Cycle

Table 7.19. *FPGA Resource Usage of 64x64 Var. Coeff. Jacobi 2D*

Parallelization Degree	BRAM (%)	DSP (%)	FF (%)	LUT (%)
1 (viv_opt)	8	2	2	4
2	8	4	4	9
4	8	9	8	19

Table 7.20. *Exe. Time Measurement Results of 128x128 Var. Coeff. Jacobi 2D*

Parallelization Degree	CPU Exe.	HLS w/o Opt.	HLS w/ vivOpt.	HLS w/ viv_poly.
1	485256	596976	401210	-
2	-	-	-	228363
4	-	-	-	142744

Unit: Clock Cycle

Table 7.21. *FPGA Resource Usage of 128x128 Var. Coeff. Jacobi 2D*

Parallelization Degree	BRAM (%)	DSP (%)	FF (%)	LUT (%)
1 (viv_opt)	25	2	2	4
2	25	4	4	9
4	25	9	8	19

Table 7.22. *Exe. Time Measurement Results of 256x256 Var. Coeff. Jacobi 2D*

Parallelization Degree	CPU Exe.	HLS w/o Opt.	HLS w/ vivOpt.	HLS w/ viv_poly.
1	1890650	2498682	15888104	-
2	-	-	-	919902
4	-	-	-	559142

Unit: Clock Cycle

Table 7.23. *FPGA Resource Usage of 256x256 Var. Coeff. Jacobi 2D*

Parallelization Degree	**BRAM (%)**	**DSP (%)**	**FF (%)**	**LUT (%)**
1 (viv_opt)	98	2	2	4
2	98	4	4	9
4	98	9	8	19

As can be understand from execution time results from tables 7.18, 7.20, 7.21, execution time of hardware accelerator is higher for all matrix sizes and all parallelization degrees because of extra memory access traffic. However, if the comparison of the offered hardware accelerator and usual HLS-based hardware accelerator performances for both variable types are considered, offered hardware accelerator has reached nearly same performance for both constant coefficient and variable coefficient Jacobi 2D iterative stencil kernel. Table 7.24 demonstrates performance of hardware accelerator with comparing acceleration ratios of constant and variable weighted Jacobi kernels for highest parallelization degrees. Acceleration ratios for constant variable Jacobi kernel is taken from table 7.17.

Table 7.24. *Performance Comparisons for Var. and Cons. Coeff Jacobi 2D*

Matrix Size	**HLS w/ vivOpt. (Var. Coeff.)**	**HLS w/ viv_poly. (Var.Coeff.)**	***Acceleration Ratio (%) (Var.Coeff.)***	***Acceleration Ratio (%) (Cons.Coeff.)***
64x64	93614	33611	179	173
128x128	401210	142744	181	177
256x256	15888104	559142	184	183

If FPGA resource usage results of experiments with constant and variable coefficient Jacobi kernels are examined, BRAM, FF, LUT usage rate for variable coefficient is higher due to extra memory requirement of weight matrix. On the other hand, DSP

usage is kept constant because of same computational intensity of both constant and variable weighted Jacobi kernels.

7.5. Complete System Evaluation

When all experimental and theoretical results in Chapter 7 are taken into consideration, it could be easily said that, Vivado HLS-based hardware accelerator design with only its inherent optimizations provide acceptable performance for computationally intense applications. However, this thesis work shows that better circuits could be designed with higher performance via combining HLS with polyhedral model-based analysis and optimization tools if there are enough hardware resources on the target SoC platform.

CHAPTER 8

CONCLUSION AND FUTURE WORKS

8.1. Conclusion

In this work, a semi-autonomous hardware accelerator for computationally intense applications is proposed. This proposed hardware accelerator is basically designed by combining three different tools. The first one is Vivado HLS, which is a high-level synthesis tool of Xilinx Company designed for their own FPGA platforms. The second one is POCC (Polyhedral Compilation Framework), which statically analyzes and optimizes nested loop kernels and the last one is PIPS, which extracts SCOP in a given source code to feed it to POCC.

In order to test and verify the effectiveness of the proposed hardware accelerator platform, different test scenarios are generated by using different size of matrix multiplications and iterative stencil loops having loop carried dependencies. Xilinx ZC702 development board, which includes Xilinx Zynq-7000 series SoC device is selected as the test platform.

To enable course grained parallelism in loop kernels, this work proposes loop tiling transformation, which is usually used in the literature for cache optimization contrary to usage in this thesis and HLS template code is generated to design parallel circuits corresponds to tiled loop kernels that is tiled automatically using POCC Tool. Unfortunately, all loop kernels are not suitable for loop tiling. In this work, polyhedral optimization method called as loop skewing is applied on loop kernels manually to enable tiling operation and the combination of loop skewing and tiling is tested and verified on ZC702 board with various types of Jacobi 2D stencil kernel.

Finally, test results in this thesis demonstrate that it is possible to increase the performance of HLS based hardware accelerator design especially for applications having huge data sets by enabling course grained parallelism in nested loop structures

if desired platform has enough FPGA resources without requiring extra working hours thanks to our semi-autonomous structure.

8.2. Future Works

The hardware accelerator that is proposed in this thesis is called as semi-autonomous. There are two main reason of why it is called as semi. Firstly, in order to synthesize nested loop structure source code with course-grained parallelism in Vivado HLS, it is required to handcraft writing of a template code. Instead of writing code by hand, there should be a loop optimization pragma in HLS compiler, for example named as "loop_blocking or loop_tiling", to allow parallelization during compilation as in the case of loop pipelining. This is one of the major drawbacks of HLS tools. Secondly, although loop skewing is a powerful loop transformation method especially for iterative stencil applications, in literature there is no polyhedral model-based source to source optimization tool having talent to apply loop skewing automatically, loop skewing should be inserted in one of the current polyhedral tools.

In this thesis work, FPGA is fully configured for only one hardware accelerator application. This work can be extended via configuring FPGA partially and dynamically to enable acceleration of various applications at the same time. Also, some minor operations between POCC and Vivado is done manually in this thesis work. This procedure can also be made autonomous while inserting tiling into HLS tool.

REFERENCES

[1] A. M. Caulfield, E. S. Chung, A. Putnam, et al., “A cloud-scale acceleration architecture”, *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp.1-13, 2016.

[2] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie and X. Zhou “DLAU: A Scalable Deep Learning Accelerator Unit on FPGA”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems vol. 36, no. 3*, pp.513-517, 2017.

[3] G.E. Moore, “Cramming More Components onto Integrated Circuits” *Proceedings of the IEEE*, pp.82-85, 1998.

[4] S.Hauck, A.DeHon, “Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation”, *San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.*, 2007.

[5] R.Nane, V.Sima, C.Pilato, et al., “A Survey and Evaluation of FPGA High-Level Synthesis Tools”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.1591-1604, 2016.

[6] J.Cong, B.Liu, S.Neuendorffer, et al. “High-Level Synthesis for FPGAs: From Prototyping to Deployment”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.473-491, 2011.

[7] Abderazek Ben Abdallah, “Advanced multicore Systems-On-Chip”. *Singapore: Springer*, 2017.

[8] M.Hubner, J.Becker, “Multiprocessor System-on-Chip Hardware Design and Tool Integration”, *New York, NY, USA: Springer-Verlag*, 2011.

[9] Intel, “Architecture Brief- What is a SoC FPGA”, *Tech. Rep,* from: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ab/ab1_soc_fpga.pdf, Accessed: December 2019.

[10] Xilinx, “Znyq-7000 All Programmable SoC”, http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, Accessed: November 2019.

[11] J.Hrica, X.Journal, “Floating-Point Design with Xilinx's Vivado HLS”, App.Note, from: https://china.xilinx.com/support/documentation/application_notes, Accessed: December 2019.

[12] V. Rana, A. Nacci, I.Beretta, et al. “Design Methods for Parallel Hardware Implementation of Multimedia Iterative Algorithms”, pp.1-6, 2011.

[13] P.Coussy and A.Morawiec, “High-Level Synthesis: From Algorithm to Digital Circuit”, *Netherland: Springer Science*, 2008.

[14] P.Coussy, D.D.Gajski, M. Meredith, et al. “An Introduction to High-Level Synthesis”, *IEEE Design & Test of Computers*, pp.8-17, 2009.

[16] A.Canis, J.Choi, M.Aldham, et al. “LegUp: An Open-source High-Level Synthesis Tool for FPGA-based Processor/Accelerator Systems”, *ACM Trans. Embed. Comput. Syst*, pp.30-33, 2013.

[17] Xilinx, “Vivado Design Suite User Guide: High-Level Synthesis*”, User Guide*, from:https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf, Accessed: December 2019.

[18] M. Fingeroff, “High-Level Synthesis Blue Book”, *Bloomington, IN, USA: Xlibris Corp.*, 2010.

[19] P.Coussy, D.D.Gajski, M.Meredith and A.Takach, “An Introduction to High-Level Synthesis”, *IEEE Design & Test of Computers*, pp.8-17, 2009.

[20] X.Liu, Y.Chen, T.Nguyen, et al. “High Level Synthesis of Complex Applications: An H.264 Video Decoder”, *ACM/SIGDA International Symposium on FPGAs*, pp.224-233, 2016

[21] C.Zhang, P.Li, G.Sun, et al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”, *FPGA 2015*, pp.35-36, 1998

[22] N.Suda, V.Chandra, G Dasika, et al. “Throughput-Optimized OpenCLbased FPGA Accelerator for Large-Scale Convolutional Neural Networks”, *ACM/SIGDA International Symposium on FPGAs*, pp.16-25, 2016.

[23] X.Gao, J.Wickerson, and G.A.Constantinides, “Automatically Optimizing the Latency, Area, and Accuracy of C Programs for High-Level Synthesis”, *ACM/SIGDA International Symposium on FPGAs*, pp.234-243, 2016.

[24] F.Winterstein, S.Bayliss, and G.A.Constantinides, “Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis”, *FCCM-IEEE 22nd Annual International Symposium*, pp.1-8, 2014.

[25] Louis-Noël Pouchet. “PolyBench/C the Polyhedral Benchmark suite”, from: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, Accessed: December 2019.

[26] C. Ancourt, B.Breusillet, F. Coelho, et al. “PIPS a workbench for intra-procedural program analyses and parallelization”, *Meeting on data parallel languages and compilers for portable parallel computing*, 1994.

[27] A.Nilsson and K.Årzén, “Static Analysis and Transformation of Dataflow Multimedia Applications”, *Technical Reports TFRT-7626-2012*, from: http://lup.lub.lu.se/record/3191921, Accessed: December 2019.

[28] K.Trifunovic, D.Nuzman, A.Cohen, et al. “Polyhedral-Model Guided Loop-Nest Auto-Vectorization”, *Parallel Architectures and Compilation Techniques Conf.*, pp.327-337, 2009.

[29] M.W.Benabderrahmane and A. Cohen. “The Polyhedral Model Is More Widely Applicable than You Think”, *International Conf. of Compiler Construction*, pp.283-303, 2010.

[30] A.Groesslinger and C.Lengauger. “Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”, *Parallel Processing Letter*, Vol.22, No.04, 2012.

[31] M.Griebl and C.Lengauer. “The Loop Parallelizer LooPo”, *LCPC*, pp.603-604, 1996.

[32] L.Pouchet. “POCC-The Polyhedral and Compiler Collection”, from: http:// web.cs.ucla. edu/~pouchet/software/pocc/, Accessed: December 2019.

[33] U.Bondhugula, A.Hartono, J.Ramanujam, et al. “PLUTO: A practical and automatic polyhedral program optimization system”, *ACM SIGPLAN Notices*, Vol.43 No.6, 2008.

[34] L.N.Pouchet, C.Bastoul, and A.Cohen, “Letsee: the legal transformation space exploratory”, pp.247-251, 2007.
[35] C´ edric Bastoul, “A Loop Generator and For Scanning Polyhedra” from: http:// bastoul.net/cloogg, Accessed: December 2019.
[36] R.M.Karp, R.E.Miller, and S.Winograd, “The Organization of Computations for Uniform Recurrence Equations”, *J. ACM 14.3*, pp.563-590, 1967.
[37] Louis-Noël Pouchet. “Polyhedral Compilation Foundations”, from: http://web .cse.ohio-state.edu/~pouchet.2/lectures/888.11.lect1.html, Accessed: December 2019.
[38] Zi Yi Ewe, “How to Find Linear (SVMs) and Quadratic Classifiers using MATLAB”, from: http://www.towardsdatascience.com, Accessed: November 2019.
[39] Paul Feautrier, “Some Efficient Solutions to the Affine Scheduling Problem: Part I. One-dimensional Time”, *Int. J. Parallel Program*, pp.313–348, 1992.
[40] A.J.Bernstein, “Analysis of Programs for Parallel Processing. Electronic Computers, *IEEE Transactions on, EC-15(5)*, pp.757–763, 1966.
[41] J.L.Hennessy and D.A.Patterson, “Computer Architecture, Fourth Edition: A Quantitative Approach” *San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.*, 2006.
[43] Paul Feautrier, “Some efficient solutions to the affine scheduling problem. Part II.Multidimensional time”, *International Journal of Parallel Programming*, pp.389–420, 1992.
[44] L.Pouchet, P.Zhang, P.Sadayappan, and J.Cong, “Polyhedral-based Data Reuse Optimization for Configurable Computing”, *ACM/SIGDA International Symposium on FPGAs*, pp.29-38, 2013.
[45] K.Underwood, “FPGAs vs. CPUs: Trends in peak floating-point performance”, *ACM/SIGDA International Symposium on FPGAs*, pp.171-180, 2014.
[46] Eric Laforest, “Survey of Loop Transformation Techniques”, from: http://fpga cpu.ca/writings/SurveyLoopTransformations.pdf2010, Accessed: December 2019.
[47] M.Kong, R.Veras and K.Stock, “When Polyhedral Transformation Meet SIMD Code Generation”, *ACM SIGPLAN Notices*, Vol.48, No.6, 2013.

[48] D.V.Rao, S.Patil, N.A.Babu, et al. “Implementation and Evaluation of Image Processing Algorithms on Reconfigurable Architecture Using C-based Hardware Descriptive Languages”, *Int. J. Theor. Appl. Comput. Sci.*, Vol.1, No.1, 2006.

[49] J. Holewinski, L.N.Pouchet, and P.Sadayappan, “High-Performance Code Generation for Stencil Computations on GPU Architectures”, *ACM ICS*, pp.311-316, 2012.

[50] Dietmar Fey, “Grid Computing”, *Nurmberg, Germany: Stringer Inc.*, 2010.

[51] Y.Liang, K.Rupnow, Y.Li, D.Min, et al. “High-Level Synthesis: Productivity, Performance, and Software Constraints”, *J. of Electrical and Computer Engineering*, pp.13-14, 2012.

[52] H.Ziegler, M.W.Hall and P.Diniz, “Compiler-Generated Communication for Pipelined FPGA Applications”, *Design Automation Conference*, pp.610-612, 2003.

[53] R.Rodrigues, J.Cardoso, and P.Diniz, “A data-driven approach for pipelining sequences of data-dependent loops”, *FCCM 2007*, pp.219-228, 2007.

[54] P.Li, L.N.Pouchet and J.Cong, “Throughput optimization for high-level synthesis using resource constraints”, *IMPACT 2014*, pp.1-3, 2014.

[55] Q.Huang, R.Lian, A.Canis, et al. “The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware”, *ACM Trans. Reconfigurable Technology and Systems*, pp.90-95, 2015.

[56] J.Cong, B.Liu, R.Prabhakar, and P.Zhang, “A Study on the Impact of Compiler Optimizations on High-Level Synthesis”, *Languages and Compilers for Parallel Computing*, pp.143-145, 2013.

[57] W.Meeus, K.V.Beeck, T.Goedemé, “An overview of today’s high-level synthesis tools”, *Design Autom. Embedded Syst.*, pp.31–51, 2012.

[58] J. Cong, P. Zhang, and Y. Zou, “Optimizing Memory Hierarchy Allocation with Loop Transformations for High-Level Synthesis”, *Design Autom. Conf. (DAC)*, pp.1233–1238, 2012.

[59] S.Bayliss, G.A.Constantinides, “Optimizing SDRAM Bandwidth for Custom FPGA Loop Accelerators”, *ACM/SIGDA Int. Symposium on FPGAs*, pp.195–204, 2012.

[60] W.Jiang, J.Cong, B.Liu, et al. “Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization”, *ACM Trans. Des. Autom. Electron*, Vol.16, No.2, 2011.

[61] Y.Wang, P.Li, P.Zhang, et al. “Memory Partitioning For Multidimensional Arrays in High-Level Synthesis”, *IEEE/ACM Design Automation Conference*, pp.12:1–12:8, 2013.

[62] C.Alias, A.Darte, and A.Plesco, “Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA”, *Proc. Conf. Design, Autom. Test Eur. (DATE)*, pp. 575–580, 2013.

[63] Xilinx, “ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC”, from: https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf, Accessed: November 2019.

[64] Xilinx, “AXI Referance Guide”, from: http://www.xilinx.com/support /Documentation/ip_documentation/ug761_axi_referance_guide.pdf, Accessed: November 2019.