POLITECNICO DI MILANO

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Elettronica, Informazione e Bioingegneria

Scuola di Ingegneria Industriale e dell'Informazione

Automated acceleration of dataflow-oriented C applications on FPGA-based systems

Relatore: Prof. Marco Domenico SANTAMBROGIO

Correlatore: Dott. Marco RABOZZI

Tesi di Laurea di:

Francesco Peverelli

matricola 876041

Anno Accademico 2019-2020

To my family and friends

FP

Ringraziamenti

Vorrei ringraziare tutti coloro che hanno reso possibile questo lavoro tramite il loro supporto, la loro guida e i loro consigli. Prima di tutto vorrei ringraziare il mio advisor, Marco Domenico Santambrogio, per la sua dedizione nel garantire ai suoi studenti opportunità di crescita e formazione personale e professionale, e per avermi permesso di sviluppare i miei interessi di ricerca che si sono concretizzati in questo lavoro di tesi. In secondo luogo vorrei ringraziare Marco Rabozzi, per avermi seguito nello sviluppo di questo progetto e per la sua disponibilità al dialogo e coinvolgimento in molte idee alla base di ciò che oggi è presente in questa tesi. Inoltre ringrazio Emanuele Del Sozzo per aver concepito questo progetto e per il contributo alla sua realizzazione. Grazie a tutto il personale e studenti del NECSTLab e del Politecnico di Milano per il gran numero di ore passate insieme, in particolare ringrazio Alberto Zeni, Qi Zhou e Guido Walter Di Donato per avermi accompagnato in questa fase della mia vita professionale e Lorenzo Di Tucci per avermi seguito su divresi progetti di ricerca. Ringrazio anche Marco Siracusa per il contributo e le discussioni riguardo nostri comuni interessi di ricerca che hanno reso possibile la realizzazione di questo progetto. Infine ringrazio la mia famiglia per il continuo supporto in questi anni di studio.

Con affetto e gratitudine,

Francesco

Contents

Abstract **XI**

Sommario **XIII**

1 Introduction **1**

1.1 FPGA overveiw 1

1.2 The role of FPGA 3

1.3 High Level Synthesis technology 4

1.4 The dataflow model 5

1.4.1 The dataflow atchitecture 6

1.4.2 Static dataflow 7

1.4.3 Synchronous dataflow 8

1.5 Problem definition 8

2 State of the art **11**

2.1 Modern HLS approaches 11

2.1.1 Vivado HLS 12

2.1.2 Bambu 12

2.1.3 LegUp 13

2.2 DSL for FPGA-based design 13

2.2.1 Darkroom 14

2.2.2 GraphStep 14

2.2.3 FROST 14

2.3 Dataflow-based design methodologies targeting FPGA 15

2.3.1 RIPL 15
2.3.2 GraphOps 16
2.3.3 Optimus 16
2.3.4 CAPH 17
2.3.5 MaxJ 18
2.4 Source-to-source optimizers targeting FPGA 19
2.4.1 Hipacc 19
2.4.2 FAST/LARA 20
2.5 Unsolved issues and proposed solution 21

3 Proposed methodology **23**
3.1 Proposed design flow 23
3.2 Code analysis and transformation 27
3.2.1 Underlying assumptions 27
3.2.2 Code normalization and analysis 28
3.3 Dataflow graph IR 39
3.3.1 DFG IR components 39
3.3.2 Transformation to DFG IR 41
3.4 DFG optimizations 44
3.5 Design space exploration 55
3.5.1 Rerolling model 57
3.5.2 Vectorization model 60
3.5.3 Optimization problem 61
3.6 Backend 62
3.6.1 Kernel generation 62
3.6.2 Manager generation 67

4 Experimental evaluation **71**
4.1 Experimental settings 71
4.2 Case studies 72

4.2.1 Sharpen filter 72
4.2.2 Asian option pricing 74
4.2.3 Quantum Monte Carlo Simulation 77
4.3 Results evaluation 79

5 Conclusions and Future work **81**
5.1 Contributions and limits 81
5.2 Future work 83

Bibliography **85**

List of Figures

1.1 Schematic example of heterogeneous FPGA structure. 2

1.2 Graphic dataflow representation of a small program. 6

3.1 Overview of the translation and optimization process. 26

3.2 Normalization passes applied to the input LLVM IR in order of application. 32

3.3 Examples of do-while and while loops as natural loops. 33

3.4 An example of natural loop normalization. In 2) we insert the pre-header, in 3) the latch and in 4) the loop exit block. 36

3.5 Detailed view of the dataflow translator component's analysis and transformations. 42

3.6 Example of a simple function and its DFG IR. 45

3.7 Example of a simple DFG IR subject to the vectorization optimization. . . 47

3.8 Example of fully unrolled sum over 3 elements with a functional unit with latency of 2 cycles. 49

3.9 Example sum over 3 elements implemented as a cyclic dataflow, with a functional unit with latency of 2 cycles. This results in a throughput of 1 sum every 6 cycles with one functional unit. 49

3.10 Example sum over 3 elements implemented as a cyclic dataflow with the data interleaving optimization and a functional unit with latency of 2 cycles. This results in a throughput of 1 sum every 3 cycles with one functional unit. 51

3.11 Latency and maximum achievable frequency for a single-precision floating-point adder (FADD) and exponential (EXP) operator for different hardware implementations on a MAX4 Galava Card. 57

4.1 The sequence of filters applied by the sharpen filter on the input image. The figure shows the filters and their data dependencies 72

4.2 The overall structure of the Asian Option Pricing application, as implemented in [51] 75

List of Tables

4.1 Summary of multiple implementations of the sharpen filter algorithm. . . 74
4.2 Summary of resource utilization for the multiple implementations of the sharpen filter algorithm shown in table 4.1. 74
4.3 Summary of multiple implementations of the asian option pricing algorithm with 30 averaging points. 74
4.4 Summary of multiple implementations of the Asian option pricing algorithm with 780 averaging points. 75
4.5 Summary of resource utilization for the multiple implementations of the Asian option pricing algorithm with 30 averaging points. 75
4.6 Summary of resource utilization for the multiple implementations of the Asian option pricing algorithm with 780 averaging points. 76
4.7 Summary of multiple implementations of the VMC algorithm. 80
4.8 Summary of resource utilization for the multiple implementations of the VMC algorithm. 80

Acronyms

ALM	Adaptive Logic Module. VII, 1
ASIC	Application Specific Integrated Circuit. VII, 4
AST	Abstract Syntax Tree. VII, 20, 29
CAD	Computer Aided Design. VII, 4
CFG	Control Flow Graph. VII
CLB	Configurable Logic Block. VII, 1
DSL	Domain Specific Language. VII, 8, 9, 11, 13
DSP	Digital Signal Processing. VII, 2, 4
EDA	Electronic Design Automation. VII, 4
FPGA	Field Programmable Gate Array. VII, XI, XIII, 1, 3, 5
GPP	General Purpose Processor. VII, 3
GPU	Graphics Processing Unit. VII, 3, 8
HDL	Hardware Description Language. VII, 4, 5, 12, 13
HLS	High Level Synthesis. VII, XI, XIII, 1, 3, 4, 5, 8, 11, 12, 13
HPC	High Performance Computing. VII, 3
LUT	Look-Up Table. VII, 1
RIPL	Rathlin Image Processing Language. VII, 15
SCEV	Scalar Evolution. VII, 32, 39, 42

SISAL Streams and Iteration in a Single Assignment Language. VII, 6

SSA Single Static Assignment. VII, 30

Abstract

The end of Dennard scaling over the last two decades has meant that computing systems could no longer achieve exponential performance improvement through higher clock frequency and transistor density due to the power wall problem. Heterogeneous computing systems address this issue by incorporating specialized hardware to achieve better energy efficiency and performance. In this context, Field Programmable Gate Arrays (FPGA) have steadily grown in popularity as hardware accelerators, although the greatest obstacle to their mainstream adoption remains the high engineering cost associated with developing FPGA-based applications. Despite the remarkable improvements in the effectiveness of third-generation High Level Synthesis tools, they still require some domain-specific knowledge and expertise to be used effectively. This thesis proposes a methodology and a tool that further increase the accessibility of HLS technology by providing a high level language frontend for the design of dataflow applications on FPGA. This framework allows software developers to write C code without focusing on FPGA-specific optimizations or concepts related to the dataflow model. The tool leverages the LLVM compiler framework to apply dataflow-specific code transformations and FPGA-related optimizations and outputs optimized code ready to be synthesized by state-of-the-art FPGA synthesis tools. A performance model tailored for dataflow computations allows obtaining accurate performance estimates before synthesis for different combinations of available optimizations. An ILP formulation of the optimization problem is then used to obtain the set of optimizations that maximizes throughput while respecting the FPGA's resources constraints. To validate this approach, we have tested the tool on different unoptimized algorithms written in C and we have targeted MaxCompiler as a backend dataflow synthesis tool. We have compared the performance obtained by these automatically optimized designs to their hand-optimized counterparts and obtained performance which ranges from 0.5x speed down to 1.34x speedup, depending on the benchmark. From the point of view of productivity, our automated optimization methodology obtains these results in about a day of work by software developers, as opposed to the several weeks of optimization by expert FPGA developers required to produce the hand-optimized designs. These results show that our methodology allows to optimize the original code and transform it into dataflow code optimized for FPGA synthesis with significantly reduced development effort.

Sommario

La fine del ridimensionamento Dennard nel corso degli ultimi vent'anni ha fatto si che i moderni microprocessori non potessano ottenere un aumento esponenziale di performance attraverso una frequenza di clock più alta e una maggiore densità di transistor. I sistemi di computazione eterogenei affrontano questo problema incorporando hardware specializzato per ottenere un miglioramento in performance ed efficienza energetica. In questo contesto, le Field Programmable Gate Arrays (FPGA) sono sempre più utilizzate come acceleratori hardware, sebbene l'ostacolo principale contro un'adozione più diffusa di questa tecnologia rimanga il proibitivo costo di sviluppo. Nonostante i notevoli miglioramenti dei tool di High Level Synthesis di terza generazione, questi richiedono comunque esperienza e una conoscienza specifica di dominio per poter essere utilizzati in maniera efficace. L'obbiettivo di questa tesi è proporre una metodologia ed un tool che migliorino l'accessibilità della tecnologia di HLS mettendo a disposizione un frontend per linguaggi di alto livello per il design di applicazioni dataflow su FPGA. Questo framework permette a sviluppatori software di scrivere codice in C senza doversi occupare di ottimizzazioni specifiche alle FPGA o concetti relativi al modello dataflow. Il tool sfrutta il compiler framework LLVM per applicare trasformazioni specifiche per computazioni dataflow e ottimizzazioni relative all'architettura target e produce come output codice ottimizzato, pronto per essere sintetizzato su FPGA da appositi tool commerciali. Un modello di performance specifico per computazioni dataflow permette di ottenere stime di risorse accurate prima della sintesi per diverse combinazioni di ottimizzazioni. Una formulzione ILP è utilizzata per risolvere il relativo problema di ottimizzazione per massimizzare il throughput rispettando le limitazioni in termini di risorse hardware dell'FPGA. Per validare il nostro approccio, abbiamo testato il tool su diversi codici non ottimizzati scritti in C e abbiamo scelto MaxCompiler come tool di backend per la sintesi del design dataflow. Abbiamo comparato le performance ottenute dai design generati attraverso il nostro tool con design ottimizzati manualmente presenti nello stato dell'arte, ottenendo performance variabili da 0.5x a 1.34x in speedup a seconda dei benchmark considerati. In termini di produttività, la metodologia di ottimizzazione automatica proposta richiede circa un giorno di lavoro da parte di uno sviluppatore software per produrre i risultati riportati, rispetto alle settimane di lavoro di ottimizazzione da parte di sviluppatori per FPGA esperti richieste per produrre i design ottimizzati manualmente. Questi risultati mostrano che la metodologia proposta permette di ottimizzare e trasformare il codice in ingresso in un codice dataflow ottimizzato per la sintesi su FPGA, riducendo notevolmente lo sforzo di sviluppo.

Chapter 1

Introduction

In this chapter, we introduce the context in which this work is developed as well as the definition of the problem we intend to tackle. In section 1.1 we introduce FPGAs and their main components. In section 1.2, we briefly discuss the role of FPGAs in modern computing. In section 1.3, we introduce HLS technology, in section 1.4 we introduce the dataflow computational paradigm and in section 1.5 we define what are the problems and limitations of modern HLS technology that we want to address via the proposed methodology.

1.1 FPGA overveiw

Field Programmable Gate Array (FPGA)s are reconfigurable integrated circuits intended for custom hardware implementation. An FPGA is generally composed of three main building blocks: Configurable Logic Block (CLB)s, also known as Adaptive Logic Module (ALM)s on Intel FPGAs, input-output blocks and communication resources. In the interest of brevity, from this point onward we will use only the terminology relative to Xilinx FPGAs, even though some architectural differences exist between different vendors. CLBs are the main components of the FPGA, used to implement either combinational or sequential logic. In Xilinx FPGAs, a single CLB is composed of a set of slices, the number of which can vary according to the device. Each slice is, in turn, composed of a set of Look-Up Table (LUT)s which store a combination of values that represent

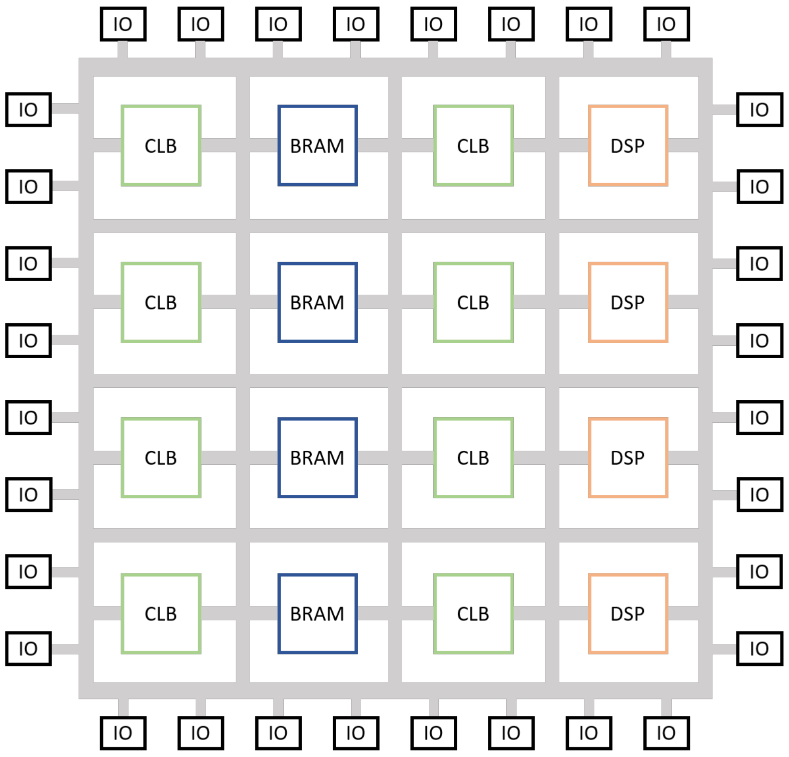


Figure 1.1: Schematic example of heterogeneous FPGA structure.

the outputs of the desired hardware function. A multiplexer reads the correct output of the function stored in a memory cell according to the given combination of input bits. Input-output blocks connect the internal logic to the I/O pins of the chip. Through their own configuration memory, IO blocks allow to configure monodirectional and bidirectional links as well as to set the voltage standards to which the pin must comply. The interconnection resources interconnect CLBs and IO blocks, creating a communication infrastructure that allows the realization of complex hardware circuits. In addition to these basic components, modern FPGAs contain other hardware components such as Block RAM cells (BRAM), processors, Digital Signal Processing (DSP) units and multipliers. Figure 1.1 shows a schematic example of a heterogeneous FPGA and its main components.

1.2 The role of FPGA

While FPGA initially flourished in networking and telecommunications, their domain of application has expanded to include embedded system applications, due to their remarkable energy efficiency, and more recently high-performance computing [1], data centers and cloud computing [2][3][4]. Traditionally, High Performance Computing was dominated by General Purpose Processors, since they were inexpensive and their performance scaled with frequency in accordance with Moore's Law. Since the mid-2000s, multicore architectures became the new way to meet the increasing demand for performance as frequency scaling was no longer a viable option, due to the escalation of power dissipation. Multicore architectures forced developers to adopt parallel programming models to fully exploit the computation capabilities of these systems. Given that multicore architectures already introduced notable complexity in traditional GPP programming, heterogeneous systems that couple GPPs with hardware accelerators such as GPUs and FPGAs became a viable alternative since they could provide great performance benefits, especially for very data-driven and compute-intensive applications [5]. In these scenarios, their specialized hardware allows to dedicate many more transistors to meaningful calculations that in GPPs are devoted to caching and memory management hardware. Over the past decade, architectural enhancements, increased logic cell count and clock frequency have made it feasible to perform massive computations on a single FPGA chip at increased compute efficiency for a lower cost. FPGA as a service has been pioneered by Amazon with its F1 instances and is a growing trend. Microsoft introduced an interconnected and configurable compute layer composed of an FPGA chip in its cloud computing environment through Project Catapult [6]. All of these factors make FPGA today one of the major players in the HPC space, as well as for embedded applications. For this reason, research surrounding FPGA development is instrumental in ensuring that this technology is used to its full potential by all types of end-users. In the next section we report a brief introduction to High Level Synthesis tools, that play a major role in the democratization of FPGAs.

1.3 High Level Synthesis technology

FPGAs were traditionally programmed through Hardware Description Languages, such as Verilog and VHDL. While these languages can be effective to program small to medium-sized, very efficient designs, the growing system complexity and the need for a shorter time-to-market for FPGA applications has created a very active field of research around CAD tools for FPGA development [7][8]. In particular, High Level Synthesis tools aim to raise the level of abstraction, allowing FPGA developers to specify their hardware design as high-level language programs. This idea of using high-level languages for hardware specification is not limited to FPGA development, as it can be used for example to design complex Application Specific Integrated Circuit (ASIC)s, but it is in FPGA development that it is most useful, since FPGA designs can be easily deployed and iteratively improved at a much lower cost compared to non-reconfigurable ASIC. HLS technology can broadly be divided into three generations, according to [7]: the first generation of tools, from the 1980s to the early 1990s, was mainly a research generation, were many foundational concepts were introduced. However, for a number of reasons these tools were a commercial failure and did not find a consistent user base. Among the reasons, [7] cites the fact that at the time RTL synthesis was just beginning to take a foothold in the community, and thus it was unlikely for behavioral synthesis to fill a design productivity gap. Moreover, these tools used little known input languages such as Silage, which represented a considerable hurdle for potential new users. Finally, the inadequate quality of results and the domain specialization of some of these tools on DSP design contributed to their limited success. The second generation, spanning from the mid-1990s to the early 2000s, saw many Electronic Design Automation (EDA) companies such as Synopsys, Cadence, and Mentor Graphics offering commercial HLS tools. Once again, the second generation was, overall, a commercial and user failure. At this point, designers who were used to RTL synthesis and were obtaining good and improving results, were not willing to change their design methodology unless HLS offered equally good or improved results with substantially lower effort and a gentle learning curve.

Second-generation HLS tools did not offer that, and instead competed in the same space by accepting behavioral HDLs as input languages, thus keeping the user base confined to RTL-level designers. In addition, these tools generally produced low-quality results for control dominated branching logic, and overall hard to validate results with high varying time intervals. The third generation of HLS tools, developed from the early 2000s until the present day, mostly use C, C++ or SystemC as input. Unlike the first two, this generation of tools is enjoying a good amount of success. Among the reasons for this success, are the fact that many of these tools focus on specific design domains, such as dataflow or DSP, and thus are able to obtain better results, and that by accepting C-like high-level languages as opposed to behavioral HDL, they effectively broaden the user base to not only expert HDL designers. Moreover, since these tools use variations of software languages, they can take full advantage of compiler optimization techniques which contribute greatly to the achievement of improved design outputs. All these advancements, coupled with the rise in popularity of FPGAs, make HLS technology a central theme in computer architectures today. In particular, the proposed methodology will focus its attention on the dataflow computational paradigm, that we introduce more in detail in the next section.

1.4 The dataflow model

Dataflow Programming is a programming paradigm whose execution model can be represented by computation nodes containing an executable block or elementary operation, having data streams as inputs and transformed data streams as output. These nodes are connected to each other forming a directed graph, which represents the entire computation. An example of a dataflow graph is shown in figure 1.2. The theoretical foundation for the dataflow programming model was first introduced by Kahn [9]. In Kahn Process Networks, the nodes are sequential processes that communicate to one another via unbounded FIFO queues. Whenever the entry FIFO queues for a node are not empty, the first values are processed and the output is sent to the FIFO belonging to the next node

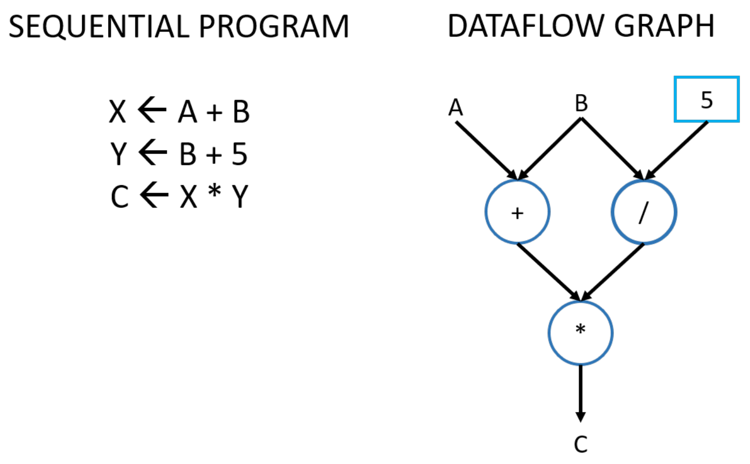


Figure 1.2: Graphic dataflow representation of a small program.

in the chain. Dataflow programming has since evolved into a methodology to exploit the capabilities for parallel processing of modern computer architectures and accelerators, as well as being the basis for several visual and text-based programming languages [10]. One of the earliest examples of dataflow programming used to exploit parallel architectures is Streams and Iteration in a Single Assignment Language (SISAL), a text-based functional and dataflow language derived from Val. It was created in the late 80s to introduce parallel computation in the first multi-core machines [11]. Many other academic dataflow programming languages have been presented since [12][13][14]. An example of commercially successful dataflow visual programming language is LabView [15].

1.4.1 The dataflow atchitecture

The reason why the dataflow execution model offered an interesting alternative to the classic von Neumann execution model comes from its inherent possibility for parallelism. In the dataflow execution model, a program begins when input data is placed on special activation nodes. When input data arrives at a set of input arcs of a node called fringe, the node becomes fireable. A node is executed at an undefined time after it has become fireable. This means that, in general, instructions are scheduled for execution as soon as the input operands become available. This model is fundamentally different from the von Neumann scheme, where a global program counter dictates which instruction will be scheduled for execution in the next cycle. In the dataflow model, multiple instructions can execute simultaneously, provided that their respective inputs are ready. Moreover, since the dataflow graph of the computation already describes the data dependencies in the program, if multiple sets of data have to be computed with the same dataflow graph, the execution of the following sets of data can begin before the first has finished executing. This technique is known as dataflow pipelining.

Despite these promising features, producing hardware implementations of the pure dataflow model has been challenging [16]. One of the sources of problems is the fact that the model makes assumptions that cannot be replicated in practice, both in terms of memory and computational resources. One of these assumptions is that the arcs connecting the nodes are FIFO queues unbounded by capacity. Since having a memory unbounded by capacity is practically unfeasible, a dataflow architecture has to rely on efficient storage techniques for storing data in the FIFOs. From the point of view of computational resources, the dataflow model assumes that any number of instructions can be executed in parallel, as long as the respective data is available. Of course, this is not practically possible, as each instruction has to be physically executed on a set of hardware resources that are finite. In order to tackle these issues, different variations of the dataflow model have been presented in the literature. In the following sections, we report some relevant examples.

1.4.2 Static dataflow

The static dataflow architecture [17] was created to address the problem of unbounded FIFO capacity. In this version of the dataflow model, each arc can hold at most one data token. A node can fire if a token is present on each input arc and no token is present in the output arcs. This check is implemented through acknowledgment signals that travel in opposite directions to each data arc and carry an acknowledgment token. In this model, the memory for each arc of the dataflow graph can be allocated at compile time and no complex hardware is needed to manage the FIFOs. However, the acknowledgment tokens increase the data traffic of the system by up to two times [18] and increase the time between successive firings of a node, which negatively impacts performance. This model also severely limits the possibility to exploit parallelism among loop iterations, often limiting the parallelism to simple pipelining [10].

1.4.3 Synchronous dataflow

Another relevant variation of the pure dataflow model is synchronous dataflow [19]. In this model, the number of tokens consumed and produced by each arc of a node is known at compile time. Due to these restrictions, only programs that can be expressed through dataflow nodes with no data-dependent control-flow can be represented. On the other hand, a program following this model can always be statically scheduled. Moreover, if a dataflow graph does not follow the synchronous dataflow restrictions, but contains subgraphs which do, it may allow partial static scheduling. Especially in domains where time is an important element of the computation, such as digital signal processing, these properties are particularly relevant.

1.5 Problem definition

Despite all the advancements that third generations HLS tools have brought to the FPGA community, FPGA development is still perceived as a big hurdle, even when compared to other heterogeneous accelerators such as GPUs. Indeed, creating an optimized FPGA design from scratch, even using modern HLS technology, requires very specific domain expertise. To achieve good performance, the developer has to either guide the HLS tool through FPGA-specific and architecture-dependent optimizations or learn to program following a completely different computing paradigm through a Domain Specific Language. An example of the former is the commercial tool Vivado HLS by Xilinx [20], where the programmer needs to insert a number of pragmas to enforce specific hardware implementations of a given portion of the program logic or handle the way in which the data is stored and partitioned in the different types of memories available on the FPGA. This and many other design decisions factor into the quality of the end result. To obtain a good quality design, the developer needs not only a detailed knowledge of the underlying architecture, but also of the specific HLS tool and its idiosyncratic behavior. An example where the synthesis tool requires the adoption of a completely different programming paradigm is MaxCompiler by Maxeler Technologies [21]. In this case, the programming language is a dataflow-specific DSL embedded in Java, which mixes traditional Java-style programming with custom variable classes and operator overloading to create a dataflow description of a given computational kernel, as well as an associated manager program. Once again the developer requires an advanced understanding of the tool-specific syntax, the dataflow computational paradigm, and the underlying architecture to produce a good design. The problem that this thesis proposes to tackle is the reduction of the gap between FPGA and software design time, by aiding the programmer with the semi-automatic optimization of C-like software functions into dataflow designs for FPGA. We have chosen the dataflow architecture as the target architectural model since it is general enough to not be limited to a single application domain while being especially proficient for data-driven and high-performance computations. Moreover, FPGAs are particularly suited for the dataflow model due to their ability to spatially distribute memory elements and functional units with customizable interconnections. Instead of trying to substitute modern FPGA synthesis tools, our framework builds on top of them, by introducing an additional frontend layer which automatically applies transformations and optimizations based on the underlying toolchain of choice, as well as the specific FPGA architecture on which the design will be synthesized. Thanks to the choice of the dataflow architectural model, we are able to evaluate the effect of the applied optimizations more precisely and therefore guide the behavior of our automated design space exploration process to deliver automatically optimized designs. A first version of this work has been published as a full paper at the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) [22], and an extension of the work has been published at the 2019 IEEE International Symposium On Field-Programmable Custom Computing Machines

(FCCM) [23] as a poster. In this thesis, we extend the work with respect to both of those publications by adding an improved design space exploration model, new optimization options and a new case study.

Chapter 2

State of the art

In the state of the art, many works have been presented with the aim of providing a more accessible way to develop FPGA-based applications. This chapter is not a comprehensive review, but presents the most relevant approaches regarding FPGA development tools. In section 2.1 we review some examples of modern general-purpose HLS tools. These tools generally take as input high-level language code and output a RTL description of the circuit to synthesize on the FPGA. In section 2.2 we report examples of DSL for FPGA development in different application domains. Differently from HLS tools, these approaches use a domain-specific input language and leverage the characteristics of a particular application domain to generate an optimized RTL design. In section 2.3 we review the subset of DSL-based approaches that generate dataflow designs. In section 2.4 we review approaches based on source-to-source code transformation and optimization that rely on existing HLS or DSL-based synthesis tools as backend. Finally, in section 2.5 we discuss how the works presented relate to the proposed methodology and highlight possible shortcomings.

2.1 Modern HLS approaches

As we described in the introduction, third-generation High Level Synthesis technology has become accepted by the FPGA community as an effective method to develop FPGA-accelerated applications with less development effort and comparable results to traditional HDL design. In the next paragraph, we report some examples of relevant HLS tools in the state of the art.

2.1.1 Vivado HLS

Vivado HLS by Xilinx [20] is a commercial High Level Synthesis tool for FPGA development which supports C, C++ and SystemC as input languages. Formerly known as AutoPilot [24], it was acquired by Xilinx and has been continually supported and improved, and is today one of the most popular FPGA HLS tools on the market. It includes a complete design environment and enables users to write, test and optimize their code by iteratively applying different optimizations through the use of FPGA-specific libraries and data types. The reports available before and after synthesis allow developers to identify bottlenecks and other optimization issues in their code and selectively optimize portions of the resulting hardware design. The tool leverages LLVM as an underlying compiler framework to extract a flexible intermediate representation and apply HLS-specific optimizations. Despite the extensive documentation, the tool is very complex and requires considerable expertise to be used effectively. Through a series of C++ libraries and optimization directives, it enables the designer to implement different computational paradigms, from master-slave to dataflow designs, with a great amount of flexibility. In addition, the designer needs to apply loop-specific optimizations, such as loop unrolling and pipelining, to extract the maximum level of parallelism from the computation, as well as choose between the different FPGA memory resources to store data. This results in a very powerful tool, which nonetheless demands a considerable development effort in order to navigate the design space and obtain a well-optimized code.

2.1.2 Bambu

Bambu [25] is an academic HLS framework that supports most C constructs. It leverages GCC to perform code optimizations such as constant propagation and loop unrolling, as well as other HLS-specific transformations. It aims at maintaining the semantics of the original application with respect to memory access and offers a highly customizable flow through an XML configuration file which enables to control, down to which algorithms should be used, the behavior of the tool. The tool outputs a HDL description of the code which can then be synthesized through specific vendors' synthesis tools.

2.1.3 LegUp

LegUp [26], an open-source HLS tool that aims to enable the use of software techniques in hardware design. To achieve this, the tool accepts standard C programs as input and automatically compiles them to a hybrid architecture containing an FPGA-based MIPS soft processor, as well as other custom hardware accelerators. LegUp is written in C++ and leverages the LLVM compiler framework for standard software compiler optimizations, and implements within the framework a custom backend for hardware synthesis. The HLS flow of the tool starts by running the program to be synthesized on the MIPS soft processor. This enables to profile the application and suggest an optimal hardware/software division for different portions of the program. At this point, the portion of the program to be accelerated by custom hardware goes through the actual HLS passes and is transformed in RTL, where standard commercial tools are used to synthesize the design.

2.2 DSL for FPGA-based design

Another trend that has been developing parallel to HLS technology for FPGA, is that of Domain Specific Languages. These languages are developed specifically for a given application domain, such as image processing, digital signal processing, and others. By leveraging the specificity of the target domain, they are able to produce very optimized hardware accelerators. The main downside of these approaches is that they force FPGA designers to learn new languages and specific syntax that is only applicable to a restricted domain. Moreover, DSLs make it harder to identify common problems among different domains, thus reducing the possibility of IP reuse. In the following section, we present some relevant FPGA DSLs in the state of the art.

2.2.1 Darkroom

Darkroom is a domain-specific language and compiler for image processing applications. The architectures targeted are ASICs and FPGAs. Based on the in-line buffering technique, Darkroom realizes very efficient hardware implementations of the specified program as an image processing pipeline. Darkroom specifies image processing algorithms as functional Directed Acyclic Graphs of local image operations. In order to efficiently target FPGAs and ASICs, the tool restricts image operations to static, fixed-size stencils. The programming model is similar to other image processing DSLs like Halide [27]. Images are specified as pure functions from 2-D coordinates to the values at those coordinates. Image functions are declared as lambda-like expressions on the image coordinates, the application of different image functions in succession creates the specification for the image processing pipeline to implement.

2.2.2 GraphStep

GraphStep [28] is a domain-specific compute model to implement algorithms that act on static irregular sparse graphs. The work presented in [29] defines a concrete programming language for GraphStep with a syntax based on Java. The language defines specific classes and functions to operate in the graph domain, such as node and edge classes and supports some atomic data types. Each of these classes supports different types of methods, such as "forward", "reduce tree" and "update" methods, which are expected to behave according to specific rules dictated by the GraphStep compute model. The framework has been tested on graph relaxation algorithms, CAD algorithms, semantic networks, and databases.

2.2.3 FROST

FROST is a unified backend that enables to target FPGA architectures. The input languages supported are Halide [27], Tensorflow [30], Julia [31] and Theano [32]. The main idea behind the framework is to provide a common intermediate representation, the FROST IR, that different DSLs can be compiled to, through an appropriate frontend. The FROST IR leverages a scheduling co-language to specify FPGA specific versions of common optimizations such as loop pipelining, loop unrolling and vectorization as well as the type of communication with the off-chip memory. In this way, FROST is able to generate C/C++ code to target FPGA HLS tools such as Vivado HLS and SDAccel. Although this approach is in part related to our methodology, it presents some key differences. Firstly, the frontend languages supported are mainly DSLs or domain-specific libraries. While a common backend to target FPGAs does increase the probability that domain experts already invested in those particular languages would consider FPGA as a possible architecture, it does not address the problem of offering an easy point of access to software developers outside of those domains. Moreover, our approach shifts the emphasis from the application domain to the computational model, thus allowing a naturally broader range of applications and uses.

2.3 Dataflow-based design methodologies targeting FPGA

In this section, we report some examples in the state of the art of languages and tools which leverage the dataflow computational model to design FPGA applications. We also give a brief description of MaxJ, a dataflow DSL for FPGA which is targeted as a backend language by the methodology proposed in this thesis.

2.3.1 RIPL

Rathlin Image Processing Language (RIPL) is a high-level image processing domain-specific language for FPGA. The RIPL language employs a dataflow intermediate representation based on a framework for describing rule-based dataflow actors [33]. The target backend for the RIPL IR is the CAL dataflow language [34], which is then compiled into Verilog. The RIPL IR supports different types of dataflow scheduling properties for its higher-level algorithmic skeletons. Some fall into the category of synchronous dataflow since all the actors produce and consume the same number of image pixels at every firing, others are categorized as cyclo-static dataflow [35]. The cyclo-static dataflow paradigm still allows for static scheduling, but also allows for actors to consume and produce amounts of tokens which vary in a cyclical pattern. RIPL showcases how the dataflow paradigm is a good fit for FPGA computation since it allows independent computational resources to operate in parallel and allows to generate hardware pipelines to hide latency.

2.3.2 GraphOps

GraphOps [36] is a modular hardware library created for the fast and efficient design of graph analytics algorithms on FPGA. Despite the fact that these algorithms are traditionally seen as fit for general-purpose architectures rather than hardware accelerators, GraphOps proposes an alternative model where a set of composable graph-specific building blocks are linked together. Graph data are streamed to and from memory in a dataflow fashion, while computation metadata are streamed through the various GraphOps blocks as inputs and outputs. In order to enhance spatial locality when accessing elements of the graph, a new graph representation optimized for coalesced memory access is also proposed. Most of the logic in the algorithms presented works well with a dataflow paradigm since feedback control is very limited. The cases where this property is violated, for example in the case of updating a global graph property for all nodes, are handled by ad-hoc control blocks.

2.3.3 Optimus

Optimus [37] is a framework designed for the implementation of streaming applications on FPGA. The input language accepted by the framework is StreamIt [38], an architecture-independent language for streaming applications. Through this language, the programmer is able to specify a series of filters interconnected to one another to form a stream graph. Stream graphs defined in StreamIt are effectively dataflow graphs that follow the synchronous dataflow paradigm. Optimus uses a specialized filter template to implement the filters specified in the input stream graph. A filter is generally composed of input FIFOs, output FIFOs, memories accessed by the filter, the filter itself and a controller. The filters are interconnected to one another by sharing the same FIFO queues. The framework allows for two different types of hardware orchestrations or modes of execution. The first is a static scheduling mode, where the compiler dictates the number of executions of each filter. In this type of scheduling, double buffering is used between pairs of filters to provide communication-computation concurrency. The other option is a greedy scheduling, where filters execute whenever data is available and are blocked upon attempting to read an empty queue. This mode of execution allows for a trade-off between the size of the queues and overall throughput. Optimus employs a variety of FPGA-specific optimizations to optimize the overall application throughput, including queue access fusion, which makes efficient use of the FPGA SRAM characteristics, and flip-flop elimination.

2.3.4 CAPH

CAPH [39] is a dataflow DSL for describing, simulating and implementing streaming applications. It is based upon two layers or levels of abstraction. The first is an Actor Description Language (ADL), used to describe the behavior of dataflow actors as a set of transition rules involving pattern matching on input values and local variables. The second is a Network Description Language (NDL), describing the structure of the dataflow networks by applying actors, interpreted as functions, to values representing wires. Contrary to similar projects, CAPH chooses a purely functional formalism to represent dataflow actors. The CAPH compiler can be used to generate a software implementation in SystemC or produce a VHDL implementation, ready to be synthesized on an FPGA. In a recent publication [40], the authors of CAPH reflect on the reception that the language has received since its release, and speculate that one of the reasons why it couldn't achieve widespread success was that it demanded from developers to abandon the traditional imperative language paradigm and adopt a completely different programming model. Moreover, the authors mention that the possibility of implementing soft-actors written in C/C++ would have been an attractive feature. In this respect, the methodology we propose aims at offering a way to implement dataflow kernels optimized for FPGA without abandoning the more common high-level programming languages and the imperative programming model.

2.3.5 MaxJ

MaxJ is a domain-specific language to design dataflow kernels for FPGA. The MaxCompiler toolchain from Maxeler Technologies [21] allows to program several dataflow kernels on a DataFlow Engine (DFE), which correspond to an FPGA, following Maxeler's Multiscale Dataflow Computing paradigm based on the synchronous dataflow model. The idea of Multiscale Dataflow Computing is to employ the dataflow model at different levels of abstraction: at a system level, multiple DFEs can be connected to form a supercomputer, at the architectural level the memory accesses are decoupled as much as possible from arithmetic operations, which are carried out with massive amounts of parallelism using deeply pipelined structures. From an architectural standpoint, a DataFlow Engine is composed of a large number of dataflow cores, simple hardware structures that carry out only one type of arithmetic computation. The data is streamed directly from memory to these dataflow cores, where the intermediate computation results flow directly from one dataflow core to another and the results are eventually streamed back to the memory. In order to achieve high throughput, DFEs make use of what in MaxJ is called Fast Memory (FMem), which refers to the BRAM blocks present on the FPGA chip, to maintain data locality and ensure that the dataflow cores have high-speed parallel access to data. Conversely, the input and output data can be streamed directly through the PCIe or from DRAM, referred to in MaxJ as Large Memory (LMem). The Maxeler Multiscale Dataflow systems can use multiple DFEs to carry out dataflow computations through a high-speed interconnect called MaxRing. From a programming standpoint, the developer needs to provide a host code, which can be written in multiple languages like C, Python, and R, that runs on CPU and performs function calls to one or more dataflow engines. The dataflow kernel specification and its attached manager program are specified in .max files, written in the MaxJ DSL. The .max files containing the dataflow kernel is a description of the computation that needs to be performed in therms of arithmetical and logical operations, whereas the manager describes the way in which the kernel is expected to interact with the host, how much data needs to be transferred and how the kernel interface relates to the host-side kernel call. MaxJ is a language embedded in Java, which makes use of custom classes, overloaded operators and proprietary libraries to effectively create a way to specify the structure of the underlying dataflow graph of a dataflow kernel. To be able to effectively program a Maxeler DFE the programmer needs to adopt a different programming paradigm, where the statements of the program actually represent interconnections between wires and hardware resources, and constructs like for loops represent a macro for hardware replication. Despite it being embedded in a high-level language, MaxJ is more akin to a high-level hardware description language. For this reason, MaxCompiler offers an effective way to program dataflow applications for FPGA experts, but the difficulties posed by the programming model and tools prevent it from becoming widespread in its adoption.

2.4 Source-to-source optimizers targeting FPGA

In this section, we list the main state-of-the-art works that use source-to-source code transformation and optimization strategies to target existing FPGA-based synthesis tools as a backend. These tools share similar objectives and strategies with the methodology proposed in this thesis, but differ from it in some respects and fail to meet some of the objectives outlined in the problem definition section of the introductory chapter, as will be discussed in section 2.5.

2.4.1 Hipacc

Hipacc [41] is a framework composed of a DSL and a source-to-source compiler for image processing. Originally created to target Nvidia and AMD GPUs, it has been extended to target FPGA [42]. The DSL is implemented through a C++ template and language-specific classes. Operator kernels are defined in a Single Program Multiple Data (SPMD) context, similar to CUDA kernels. Custom operators are defined by inheriting the Kernel class defined by the framework and overriding the appropriate methods. DSL-specific pragmas are introduced to enable customized bit-widths, The source code is compiled into an Abstract Syntax Tree through the LLVM compiler framework, using Clang as a C++ frontend. After applying optimizations appropriate for image filters and vendor-specific transformations, the tool generates code that is later synthesized by HLS tools such as Vivado HLS.

2.4.2 FAST/LARA

A framework that aims to solve a similar problem to the ones laid out in the problem definition section of the introduction is FAST/LARA [43]. Specifically, the problems identified by the authors are the need for an intuitive, concise and well-understood language to specify dataflow designs and parametrizable optimization strategies that allow design space exploration and code reuse. The frontend language proposed is FAST, which is based on C99 syntax with the addition of some unique APIs to express dataflow computations. In order to specify the possible optimizations applicable to the code, the FAST language is coupled with LARA [44], an aspect-oriented language for embedded systems which enables to select compiler optimization strategies for specific portions of the code. The compilation backend of the framework is MaxCompiler. After the first compilation, the feedback from the backend compilation process is used to drive the successive design space exploration, until certain user-specified requirements are met. The aspect-driven optimization strategy relies on different types of aspects: system aspects, which capture system-level optimizations such as software/hardware partitioning, implementation aspects, which focus on low-level optimizations such as operators optimization, and development aspects, which capture transformations that have an impact on the development process such as debugging. Using LARA the user can implement and combine these aspects to enable systematic design space exploration of all the optimization options exposed by FAST.

2.5 Unsolved issues and proposed solution

The main issue that prevents modern HLS technology from finding mainstream adoption among developers outside of FPGA experts, stems from the fact that these tools provide a variety of design and optimization options which are necessary for these tools to deal effectively with the complexities of hardware synthesis but force the users to develop a strong expertise and in-depth knowledge of the tool and architecture. On the other hand, DSL-based approaches reduce the complexity of the process by focusing on a particular application domain and by introducing a language that is designed to restrict the FPGA design space so that more specific optimizations become possible. The main drawbacks of these approaches are that the users need to learn how to program in very niche languages which are often limited in their use to a particular tool or application domain, and in the case of dataflow-based DSL learn to program in a completely different programming paradigm from the more common imperative languages. Our solution aims to solve these limitations by accepting standard C code as a frontend language and deals with the complexity of the design space by restricting the types of target designs to dataflow-based accelerators, which allows for an effective automatic optimization process. The tools presented in section 2.4 have some similarities to the approach proposed in this thesis but differ in some key aspects. Despite implementing source-to-source compilation to automatically optimize the code, Hipacc restricts the domain of application to image processing kernels and automatically applies optimizations specific to this domain. Our approach is more general since it applies different optimizations in order to optimize the specific dataflow function specified in terms of throughput and hardware resources used, regardless of its domain of application. Moreover, Hipacc is compiled through a DSL, which inherently restricts the usability of the tool by non-expert designers. Despite the usefulness of presenting a more familiar language as a frontend to incentivize developers to explore the possibilities offered by FPGA dataflow designs, the FAST/LARA framework introduces several complications that simply move the problem elsewhere: the LARA aspect-oriented language is possibly just as unfamiliar to the vast majority of software developers and the need to program FPGA-specific optimizations through the use of compilation pragmas, although similar to already existing HLS approaches such as the one offered by Vivado HLS, does not diminish the amount of FPGA-specific expertise necessary to develop an optimized application. In the following chapter, we outline more in detail how the proposed methodology deals with these limitations to increase the productivity of FPGA-based design.

Chapter 3

Proposed methodology

In the following chapter, we outline the proposed methodology and describe in detail how the tool we have implemented allows a non-expert user to obtain an optimized dataflow kernel synthesizable on an FPGA starting from high-level C code. In section 3.1 we describe the overall development flow and all the major components of the framework, in section 3.2 we describe how the input code is analyzed and which transformations are applied in order to obtain a dataflow representation of the computation and section 3.3 describes the dataflow intermediate representation that is used to apply all the available optimizations. Section 3.4 describes those optimizations in detail, section 3.5 outlines the semi-automated design space exploration process and section 3.6 describes the backend code generation phase.

3.1 Proposed design flow

This section describes the overall design flow of the methodology proposed in this thesis and the main components of our framework. A summary of this design flow is depicted in figure 3.1. The framework we present takes as input a C source file, containing a function we want to transform into a dataflow kernel. We start from the assumption that the user has identified that part of the code as a bottleneck in a particular program, and wants to build a dataflow accelerator for that function. This assumption is reasonable, in the sense that the use of profilers such as perf [45] or valgrind [46] to identify computational bottlenecks is not uncommon among software developers. We imagine the typical user of our framework as an expert developer who is concerned with performance improvement on a specific application and is aware of the possibility of hardware acceleration but does not possess the expertise or the resources, in terms of development time, to optimize by hand an accelerator himself.

Once the portion of the program to accelerate has been extracted into a function, the user can insert it into a simple template we provide to start the code transformation process. Internally, we obtain the LLVM IR of the function from our template by calling Clang [47], the C language frontend for LLVM. Alternatively, the user can directly invoke the code transformation tool from the command line on a .ll file, which is an LLVM IR module obtained through an LLVM frontend compiler, and specify the name of the function he/she wants to accelerate. Optionally, the user can decide to specify extra arguments representing a combination of available optimizations. If this is the case, the tool will produce the specified version of the kernel as a starting point for the design space exploration process. If no arguments are specified, the kernel is first transformed into a non-optimized version, and the later design space exploration will inform the tool on the optimal combination of optimizations to apply. The second step of the proposed methodology involves a series of code normalization steps and analysis steps, some of which are existing LLVM passes and some are custom passes implemented for our specific needs. A complete list and detailed explanation of these analyses and transformations is available in section 3.2. The purpose of this normalization phase is two-fold: onto one hand, we want to apply all possible transformations to the code that, while preserving semantical equivalency, make it easier for the following transformations steps to get from an imperative language like C to a dataflow representation of our function. On the other end, with some simple analysis steps, we check if any characteristics of the input code prevent it from being successfully translated into our backend language. In fact, while our tool accepts most constructs of the C language, at present it requires the code to have some characteristics, for example, no recursive function calls, to ensure a correct translation. Some of these requirements are not necessary from a theoretical standpoint, but lifting them would require a good amount of development effort for a comparatively small gain in the expressiveness allowed in the code. From a methodological standpoint, most of these do not represent a real limitation of our approach and could be integrated into the tool at a later stage of development. After the normalization passes have been applied and the function as been deemed fit to be translated into a dataflow kernel, the imperative code is transformed into a dataflow graph intermediate representation. This representation introduces an important element of flexibility in our methodology. At this stage of the translation process, the code has already been transformed from a sequence of imperative statements into a dataflow representation of the computation, where each node represents an independent computational resource, and the data flows from one node to the other to compute the expected output. Nonetheless, this representation is still agnostic to all the implementation-related and architectural constraints, therefore it can be used to target different dataflow backend languages. Once we have a first dataflow graph representation of the function, the design space exploration module analyzes the graph to identify the optimal combination from the available optimizations that maximizes the overall throughput without exceeding the hardware resources available on the FPGA. All the optimizations are applied to the dataflow graph and its metadata, to obtain the final dataflow representation that will be translated into the target backend language. In this phase, we start to introduce architectural elements and hardware constraints into the optimization process which are essential to obtain a working and optimized design. To maintain as much decoupling as possible between architecture-agnostic transformations and implementation details, the domain space exploration module uses a target-dependent resource estimator, that traverses the dataflow graph and estimates the amount of hardware resources required for its implementation, to infer the impact on hardware resources of code transformation that ultimately result in hardware replication, insertion of memory elements for parallel data access and increase or decrease of latency for the design. Together with target-independent transformations, the design space exploration process includes some target-dependent optimizations that leverage different possible implementations of hardware modules, the amount of pipelin-

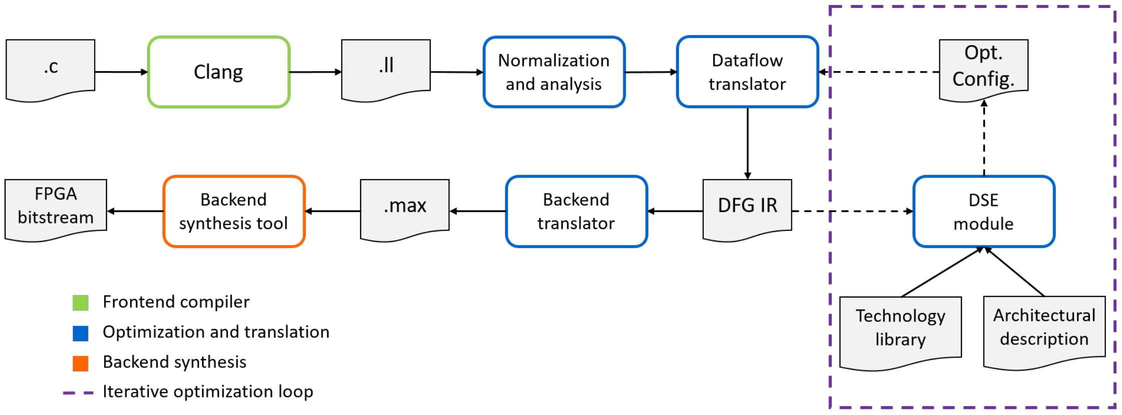


Figure 3.1: Steps of the translation and optimization process. After a first compilation step with Clang, our framework takes as input the LLVM IR source file and an initial optimization configuration and produces a dataflow intermediate representation of the design. A technological library that contains empirical data on the available hardware resources and the dataflow IR are taken as input by an optimizer that generates the optimal configuration of parameters which is used to generate the final synthesis-ready code.

ing for functional units and some degree of control over synthesis frequency. After all the optimizations have been applied, the final version of the dataflow graph is given to the backend translation module, which produces as output the optimized version of the code. To verify the results of the proposed approach, we have selected MaxCompiler [21] as a backend synthesis tool. Our framework produces two MaxJ source files containing the implementation of the dataflow kernel and its related manager. The framework also produces a test host code that is used to simulate the design and perform a basic check for semantical equivalency between the software and hardware implementations over a randomly generated set of input values within specified ranges. Once the user has verified if the generated design satisfies their needs, he/she can use the backend toolchain to synthesize the design and deploy it to an FPGA device.

3.2 Code analysis and transformation

In this section we discuss in detail the analysis and transformation applied to the input code to get from an imperative description of the computation to a dataflow representation. In the first paragraph, we discuss the assumptions that we make on the set of input functions that our framework is able to translate, some of which are necessary for the subsequent set of transformations. In the second paragraph, we discuss the analysis and normalization steps applied, how they affect the code and their utility in transitioning to the dataflow IR.

3.2.1 Underlying assumptions

In terms of overall code structure, we assume that the computation we want to accelerate can be structured as a loop over a given, generally large, amount of iterations, that can contain other arbitrarily nested loop structures, function calls, arithmetical and logical operations. While this assumption is not a trivial restriction on the type of functions that are suited for translation through the proposed methodology, it reflects the role that the proposed framework is intended to fulfill: this methodology is not intended as a general way to translate any type of computation to a dataflow architecture. There are several reasons for this choice, firstly, the problem of transforming any type of program that can be expressed in C as a dataflow kernel is too general, and the results obtained by such an effort risk being too hard to optimize, resulting in a failure similar to the one of first and second-generation HLS technology. Moreover, structuring the input code as an iteration over a large amount of data is precisely where a dataflow accelerator can obtain good results, since it is, in essence, a complex pipelined structure for processing data in a streaming fashion, and if the code can not be conceptualized in a similar way, perhaps the developer should explore different means of acceleration. All the subsequent analyses rely on this assumption since the iterations of this outer loop are interpreted as ticks for our synchronous dataflow accelerator. This does not mean that the ticks coincide with physical clock cycles on the hardware device, or that the kernel ticks and outer loop iterations are in a 1:1 correspondence in the final translation since several factors can intervene in changing the pace at which input data is read and the latency of the different portions of the hardware pipeline. The second restriction we put on the input code is that the outer loop should not have any loop-carried dependencies, outside from reading or writing variables accessed through pointers passed as arguments to the accelerated function. This is not a restriction that has to do with the proposed methodology but is one of the current limitations of the tool. A third assumption regards the absence of recursive function calls since these are generally not well-suited for hardware acceleration. If possible, the user should modify the algorithm to perform the same computation iteratively. The fourth assumption on the input code is that all the loops present in the function with a nesting level greater than 1 have a bounded number of iterations. Lastly, we assume that the input parameters of the functions passed as pointers do not alias, both in terms of the initial memory address passed to the function, and all subsequent accesses performed through pointer arithmetic form that base address. This assumption is not unreasonable, since the use of different pointers to access the same memory location is not generally advised, and can be remedied by writing the code to abide by this rule. The reason behind this assumption is that memory accesses performed on the pointer arguments of the function through the iteration variable of the outermost loop will be interpreted as reading elements from a stream of data or writing to it, and streams are assumed to be non-overlapping and identified uniquely through the corresponding pointers in the original code.

3.2.2 Code normalization and analysis

The proposed tool operates within the LLVM compiler framework, and makes use of many analysis and transformation passes already implemented within it, as well as some custom passes. The LLVM framework provides an array of tools that allow building upon existing compiler technology and theoretical knowledge and is particularly suited for extensions and implementation of new tools due to its very modular nature. In the next paragraph, we give a brief introduction to LLVM and the LLVM IR and define some concepts that will be mentioned later in the text.

Introduction to LLVM

Traditional static compilers present a modular structure composed of three main components: a frontend, an optimizer, and a backend. The frontend deals with language-specific lexical analysis and parsing, as well as statements and data structures lowering. It transforms the code in an Abstract Syntax Tree (AST), which can be converted to another intermediate representation for optimization. The optimizer is usually mostly independent from the source language and target and performs a broad variety of analyses and transformations to improve the code running time and eliminate redundancy. The backend is responsible for mapping the code onto the target instruction set. In doing this, the objective is to generate not only correct code, but good code that takes advantage of the features offered by the target architecture. The main advantage of the three-phase compiler design is its flexibility: the compiler is able to support different frontends, so long as they all produce the same intermediate representation, as well as multiple backends for different architectures while maintaining the same core optimizer. Although this three-phase approach is well accepted in theory, it is very hard to fully realize. The LLVM compiler framework represents a good example of how this type of design can facilitate further developments in compiler technology, having given birth to numerous sub-projects as well as a number of independent projects that use LLVM as a starting point. In the case of this work, we relied on the LLVM C frontend compiler, Clang, to provide a reliable frontend for the C language. In terms of the optimizer, our methodology takes advantage of the features presented by the LLVM IR and some optimization options, aiming towards compatibility with multiple frontends which support the LLVM IR as output, and presents a custom dataflow intermediate representation which builds upon the features of the LLVM IR to provide a similar starting point for dataflow-specific optimizations. This intermediate representation provides an opportunity to target different backend synthesis tools by simply substituting the code generation portion of our tool. The LLVM IR is a low level RISC-like virtual instruction set in three address form. It allows linear sequences of simple instructions, like add, sub, compare and branch, as well as some high-level instructions like *call* and *ret*, which abstract away calling conventions and *getelementptr*, used to handle pointer arithmetic. The LLVM IR is strongly typed, with primitive types like $i32$ and $float$, and pointer types like $i32*$. Functions and statements are also typed. The LLVM IR is Single Static Assignment (SSA)-based, allowing for more efficient optimizations. This means that in each function, for each variable $\%foo$, only one statement exists in the form $\%foo = ...$, which is the static definition of $\%foo$. In the code snippet below we show two simple functions which add two numbers, one in a straightforward way and the other recursively, and their corresponding translation into LLVM IR.

```
unsigned foo(unsigned a, unsigned b) {
  return a+b;
}

unsigned bar(unsigned a, unsigned b) {
  if (a == 0) return b;
  return bar(a-1, b+1);
}
```

```
define i32 @foo(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @bar(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @bar(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

Other than being implemented as a language, the LLVM IR is also defined in three isomorphic forms: the textual format corresponding to the .ll file extension which is the one shown above, an in-memory data structure and a bytecode format, an efficient on-disk binary format corresponding to the .bc file extension. The LLVM optimizer operates on the in-memory IR representation through a series of steps called passes. A pass is a pipeline stage that operates a particular code analysis, usually producing useful metadata for subsequent passes, or some code transformation. An LLVM pass can inherit different pass interfaces depending on the granularity and type of the transformation or analysis it performs, that can operate at the module level, function level or basic block level. It also allows to specify the dependencies of the pass, in terms of transformations and analysis it requires, and its effect on the IR in terms of analysis and code properties it preserves or invalidates. These features allow writing passes in a completely modular fashion while ensuring that a given pass can clearly specify the preconditions necessary for it to perform its intended functionality. In the following paragraph, we report firstly the main analysis passes on which our tool relies, then the full list of LLVM transformation passes applied during the normalization process, describing their function and their purpose in this context. Figure 3.2 reports the sequence of passes in order of application. The version of LLVM employed is LLVM 4.0. The input for the normalization process is the LLVM IR for the function that we want to accelerate, compiled without any initial optimization.

Dominator Tree analysis

This analysis implements in LLVM the concepts of dominators and dominator tree in the CFG. These concepts are very commonly used in compilation theory, here we report the

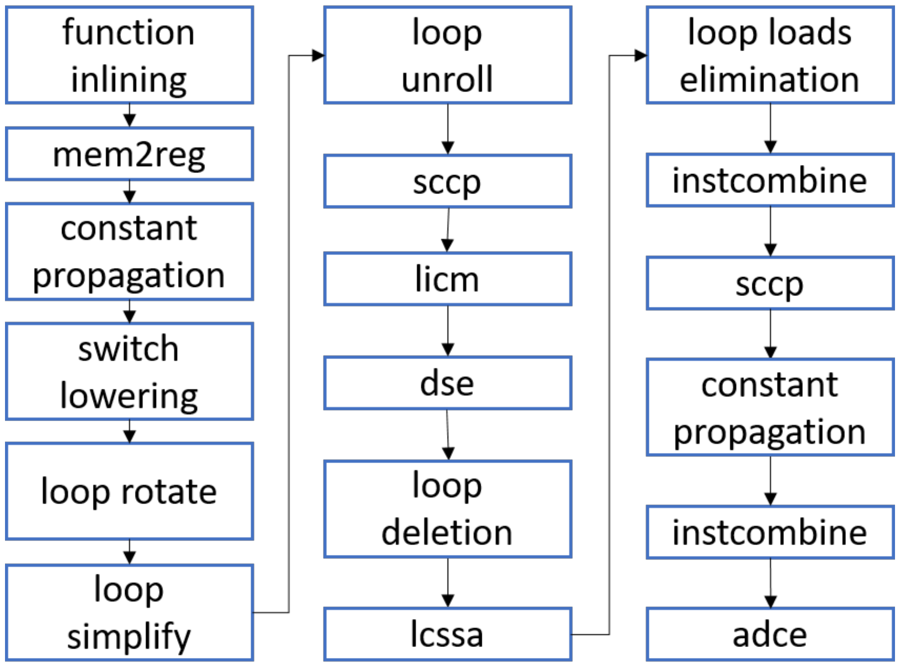


Figure 3.2: Normalization passes applied to the input LLVM IR in order of application.

definitions of these terms. In control flow graphs, a node d dominates a node n if every path from the entry node to n must go through d. A node d strictly dominates a node n if d dominates n and $d \neq n$. The immediate dominator of a node n is the unique node that strictly dominates n but does not strictly dominate any other node that strictly dominates n. Every node, except the entry node, has an immediate dominator. A dominator tree is a tree where the children of each node are those nodes it immediately dominates. Because the immediate dominator is unique, it is a tree. The dominator tree analysis of LLVM exposes an interface to inquire about whether a particular basic block in the CFG dominates, or strictly dominates another and other related questions.

Scalar Evolution analysis

The LLVM Scalar Evolution (SCEV) analysis carries important information on the evolution of scalar values, which in this case refers to single variables or constants, across different loop iterations. The SCEV analysis pass exposes an interface that maps values

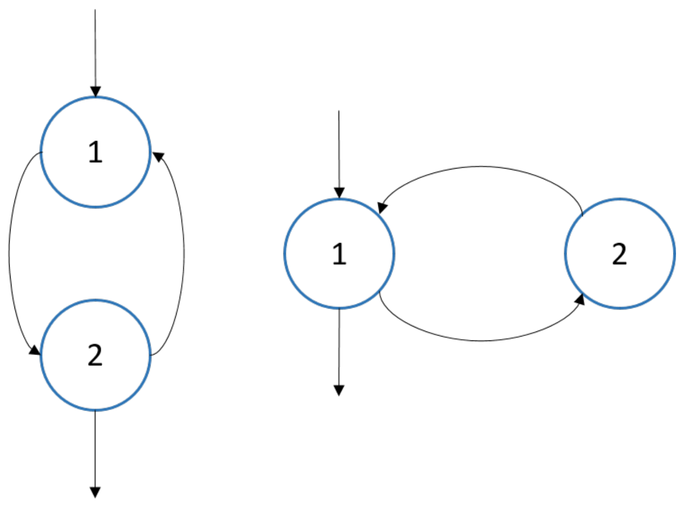


Figure 3.3: Examples of do-while and while loops as natural loops.

in the LLVM IR to SCEV expressions. These expressions form a three-like symbolic representation of the evolution of each value across loop iterations, in terms of the operators that are applied to it and the other variables involved in its evolution. Moreover, the SCEV analysis allows to infer the trip count of loops and whether this value is a constant determined at compile-time.

Loop Info analysis

This analysis pass provides an interface to inquire about the nesting structure of an LLVM IR function. The analysis identifies loops in the function and their hierarchical organization, and associates each basic block contained in a loop structure to the corresponding loop metadata. LLVM recognizes as loops only natural loops, which are defined as having only one entry node, called header, and a backedge that enters the loop header. Figure 3.3 shows examples of natural loops. This analysis does not identify as loops complex loops nor all the strongly connected components in the CFG. Moreover, this pass calculates on the fly information on whether a loop has a pre-header, the number of backedges to the header, the successor blocks of the loop and many others.

Function inlining

We implemented a custom function inlining policy based on our assumption of non-recursion and the target-dependent technology library. Whenever possible, we try to inline each function call to allow further constant propagation and optimization. In this case, we are not concerned with code size or modularity at the function level, since our primary objective is ultimately to construct a complete dataflow graph of the computation, to later apply cost-saving optimizations if the FPGA resources are scarce. The main exception to this policy are mathematical functions like *exp*() from ”*math.h*”, for which the target synthesis tool offers a specific hardware implementation. The functions to be inlined as marked by our pass with the ’alwaysinline’ attribute, then we run the LLVM *AlwaysInlinerLegacyPass* to perform the actual inlining procedure.

Memory to register pass

This is a standard LLVM pass that promotes memory references to be register references. In the LLVM IR, the *alloca* instruction is used to allocate memory on the stack frame of the currently executing function and is automatically released when the function returns. To express reading a value from memory the IR uses a *load* instruction, and to write a value to memory it uses a *store* instruction. If an *alloca* instruction only has *load* and *store* as uses, the instruction is eliminated and promoted to a register. We define the uses of a given IR instruction I as the set of instructions S such that the variable defined by I appears as one of the operands for each s in S. The *alloca* instruction is transformed by using dominator frontiers to place ϕ nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. A ϕ node is an instruction used to specify which version of a variable should be chosen at a confluence point, depending on the preceding basic block. This transformation pass is essential to enable further optimizations.

Constant Propagation

We use the LLVM *ConstantProp* pass to perform simple constant propagation. It replaces instructions that contain constant values as operands with the result. For example, it substitutes all the uses of instructions like $\%foo = add\ i32\ 1, 5$ with the corresponding result, the constant $i32\ 6$ in this example, and eliminates the instruction.

Switch lowering

The LLVM IR has a *switch* instruction as a generalization of the *br*, or branch, instruction, allowing a branch to occur to one of many possible destinations. Since at present our tool does not support this construct, we use the LLVM *SwitchLoweringPass* to transform switches into simple branches.

Loop rotate

We implemented a custom loop rotation pass to allow safe loop-invariant code motion. Loop rotation is a generalized version of the loop inversion transformation, which essentially transforms while loops into do-while loops. To maintain semantical equivalence after the transformation, the resulting do-while loop is wrapped with a conditional to ensure that the loop does not execute one iteration when the entry condition of the original while loop would have been false. The custom rotation pass we implemented performs loop rotation on all those loops for which the resulting conditional can be removed, since we do not want to deal with loops nested into conditionals. In particular, loop rotation will always be applied to all the loops at nesting depth greater than one, since we assume they perform a known number of iterations. The trivial conditional wrapping the loop is guaranteed to be eliminated through sparse conditional constant propagation.

Loop simplify

The loop simplify pass normalizes natural loops, making them more optimization friendly. This normalization process involves the insertion of a loop pre-header, which ensures that

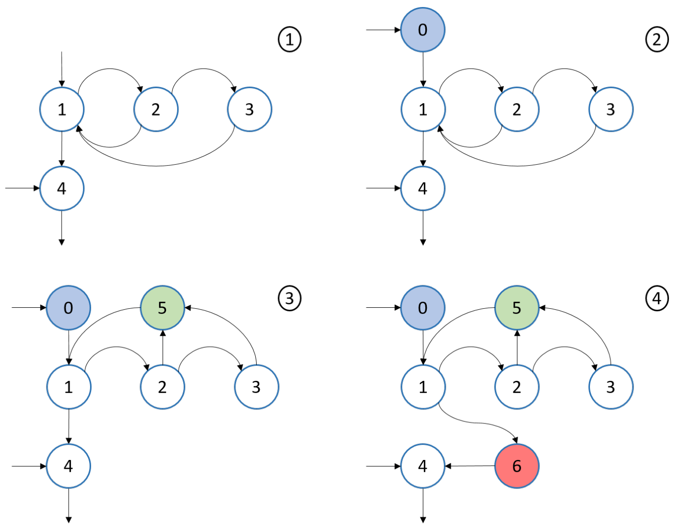


Figure 3.4: An example of natural loop normalization. In 2) we insert the pre-header, in 3) the latch and in 4) the loop exit block.

the header basic block has only one predecessor, a loop latch, which becomes the source basic block of the only backedge, and an exit block which ensures that the exit from the loop is always dominated by the loop header. In figure 3.4 we illustrate an example of this loop normalization procedure.

Loop unroll

In this pass we use the loop unroll function provided by LLVM to completely unroll all the loops with nesting depth greater than three since our current tool does not handle well more than two nesting levels. This is possible due to our assumption of knowing the number of iterations performed on these loops at compile time. This optimization strategy has worked well for our methodology so far, although it is possible that on particular code examples this may not be optimal. In the future, implementing a custom cost-evaluation for loop unrolling may be a good improvement to the current strategy.

Sparse conditional constant propagation

SCCP is a standard LLVM pass that implements sparse conditional constant propagation and merging. This pass assumes values to be constant unless proven otherwise, and basic blocks to be dead unless proven otherwise. Then, it replaces the values proven constant with the appropriate constant values, and it proves conditional branches to be unconditional.

Loop invariant code motion

This standard LLVM pass performs loop-invariant code motion. This is done by either hoisting code into the pre-header block or sinking code to the exit block. This pass also eliminates loads and stores which must alias in the loop, by promoting the corresponding variables to live in registers. If there is a store instruction inside of the loop, the pass tries to move the store after the loop. This can only happen if the following conditions are met: the pointer stored is loop invariant, and there are no stores or loads in the loop which may alias the pointer. Moreover, there are no calls in the loop which modify or reference the pointer. If these conditions are true, the pass promotes the loads and stores in the loop of the pointer to use a temporary variable whose space is allocated through a temporary *alloca* instruction. Later, the pass constructs the appropriate SSA form for the value.

Dead store elimination

This standard LLVM pass implements a simple elimination of dead store instructions that only considers basic-block-local redundant stores that may have been created by previous optimization passes.

Loop deletion

This standard LLVM performs dead code elimination for those non-infinite loops that can be proven dead. We use this pass to eliminate loops that may have been left dangling after hoisting instructions

Loop-closed SSA form

This standard LLVM pass is a loop transformation pass that inserts ϕ nodes at the end of loops for each value that is live across the loop boundary. The main usefulness of this pass is that it makes further loop optimizations simpler. In our case, it also simplifies the translation process for nested loops.

Loop loads elimination

This standard LLVM pass implements a loop-aware load elimination procedure. It uses the LLVM *LoopAccessAnalysis* to identify loop-carried dependencies with a distance of one between stores and loads. The source value of the store is then propagated to the user of the corresponding load, making the load dead. In our case, this pass is particularly useful to eliminate trivial store-load pairs which may have been created after the unrolling procedure.

Instruction combine

This LLVM pass is used to perform algebraic simplification and combine instructions into fewer simpler instructions. In our case, is particularly useful to eliminate trivial ϕ nodes that may have been created by the application of the LCSSA pass.

Aggressive dead code elimination

This LLVM pass assumes that all instructions are dead until proven otherwise, and this allows to eliminate some dead code, particularly involving loop computation. We use this pass after another application of the sparse conditional constant propagation pass, normal constant propagation pass, and instruction combine pass since this gives us the possibility to eliminate some additional branching in the code.

3.3 Dataflow graph IR

In this section, we present the dataflow graph intermediate representation that is used to express and optimize the dataflow computations in our tool. First, we introduce the main elements of this IR and then we discuss how we can transform the code from an imperative representation like the LLVM IR into a dataflow graph.

3.3.1 DFG IR components

The dataflow intermediate representation we propose comes in the form of one or more in-memory directed graphs representing the computation. In this representation, we have converted the function to accelerate into a streaming computation, thus we assume that input data will be streamed to the FPGA device, where the input data streams will be processed by a series of dataflow functional units and will produce one or more output streams that will be streamed back to the host. The elements of our dataflow graph IR are organized in different classes of dataflow functional units, which correspond to nodes of the graph.

Input stream nodes

The nodes of the graph with no predecessors can be either input stream nodes, constant values or read-only memory elements. Input stream nodes represent entry points for streams input of data. They are created when the computation is transformed into a dataflow computation by inferring that a particular load operation associated with the iteration variable of the outermost loop reads at each cycle from the base address of a pointer passed as an argument to the function to accelerate. This process will be explained in detail in section 3.3.2. Offset nodes can be placed after an input stream node to modify the way in which elements are read from a particular stream according to the Scalar Evolution (SCEV) expression associated with the iteration variable of the loop.

ROM nodes

Read-Only Memory (ROM) nodes allow us to include in the design memory elements that can be used to read constant data know at compile-time, without the need to create an artificial input stream. These elements are useful to translate read-only global data structures or function-local temporary variables, allowing us to exploit the fast parallel data access of the FPGA's RAM blocks.

Operational nodes

The intermediate nodes in the graph are nodes which generally express elementary arithmetic or logical operations. These nodes have as operands their predecessors in the graph, and the result of the operation is used by the successor nodes.

Output stream nodes

The nodes of the graph with no successors are output stream nodes. It's important to note that in order to correctly translate the computation, all the side effects that the execution of the function may cause must be considered as output streams, regardless of whether they are effectively a stream of values generated at each tick of the kernel or a single store operation. In general, the computation will have several store operations that modify either the value of a memory location whose base address pointer has been passed as a function argument or a global variable. These operations with side effects will be interpreted as output stream nodes.

Loop nodes

The graph allows for hierarchically nested loop structures so that we can identify and optimize nested computations independently from the rest of the graph, and also identify ways to synchronize how these nested structures consume and produce data. For this purpose, our representation uses the concept of loop nodes, which generally come from a nested computation in the original imperative code, and represent a portion of the graph that can contain feedback arcs. Loop nodes contain a dataflow sub-graph, which describes the nested computation. These nodes generally have multiple predecessors and successors, which include all the values used by the dataflow sub-graph of the node and all the external nodes which use some value generated within the loop node boundary. The loop node internally stores all the specific data dependencies from and to external nodes, thus maintaining both a coherent high-level view of the graph for a given nesting level as well as a complete description of the data dependencies in the computation.

Function nodes

The function calls that were not inlined during the normalization process, are treated as single nodes that have the nodes corresponding to their argument list as predecessors and as successors all the nodes which use the return value of the function call. In the backend, these functions will be implemented according to the implementation available to the backend synthesis tool.

3.3.2 Transformation to DFG IR

The most crucial step in the translation process of our tool is the transition from the LLVM imperative intermediate representation to our custom dataflow IR. An example of this transition is shown in figure 3.6. The figure shows the input C code for a function to accelerate and its corresponding DFG intermediate representation in graphical form. To perform this transition our tool relies on the assumptions that have been described in section 3.2.1. These assumptions are checked as the first step of the translation process. If any of the assumptions are not verified, the tool terminates with an error message. The dataflow translation portion of our tool can be divided into three main components: identification of input and output streams, dataflow graph construction and loop nodes construction. After these target-independent transformations, we perform two target-dependent analyses that enable us to carry out the automatic design space exploration, namely a resource estimation analysis and a graph latency analysis. A schematic view of these components is shown in figure 3.5.

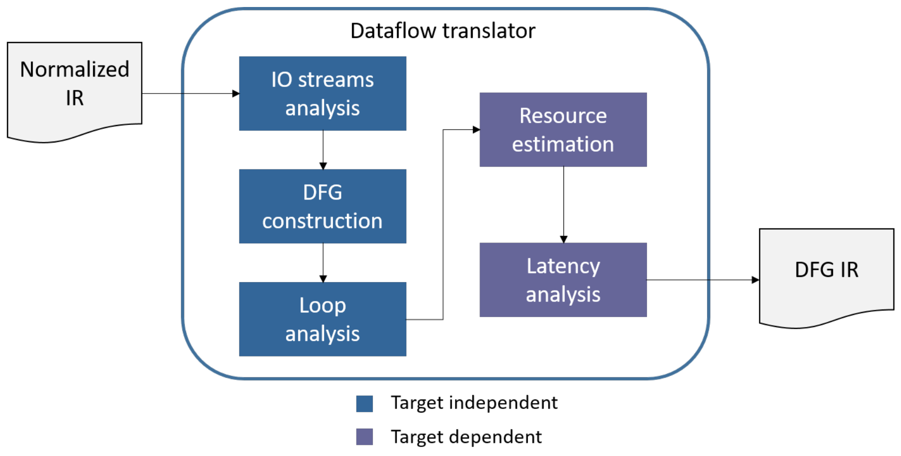


Figure 3.5: Detailed view of the dataflow translator component's analysis and transformations.

Identification of IO streams

The first step consists in the identification of the input and output streams for the function. Since the computation is guaranteed to be contained in a loop, this allows us to eliminate the loop and semantically render each iteration as one or multiple ticks of our dataflow design. We use the SCEV analysis provided by LLVM to analyze how the loop iteration variable is used to read or write memory within the function. For each memory address which is accessed through an LLVM *getelementptr* instruction, we check if it uses, directly or through a chain of uses, the induction variable of the loop. If this is the case, we check if these accesses are then used by a *store* instruction. If the store creates side effects outside the function, for example, if the base address has been passed as a function argument, we identify that memory location as an output stream. To identify input streams, we iterate over the remaining function arguments and identify through SCEV analysis those pointers which are used to access memory via the outer loop's induction variable, similarly to the ones we identified as output streams, but whose users are never store instructions.

Graph construction

Starting from the store instructions, we construct dataflow sub-graphs for which we can prove a direct dataflow relation between nodes. This is the case for example for a chain of arithmetical or logical operations which ends with a store instruction. Typically, these dataflow sub-graphs stop either at a loop boundary in the control flow graph or with operations that have constants or values loaded from memory as operands. The dataflow relations between instructions across loop boundaries are handled in the loop analysis phase, therefore, for this first construction phase, we are dealing with graphs without considering relations across different nesting depths. After constructing these sub-graphs by assigning the corresponding node types to each instruction, we need to verify if we can prove more dataflow relations among sub-graphs. This relies on the following consideration: given a loop containing a store instruction s and a load instruction l, there is a Read After Write (RAW) dependency if, at loop iteration i, the memory location read by l has already been written by s at an iteration $j <= i$. Through SCEV analysis, we can verify this condition and connect the corresponding nodes of the graph. If the RAW dependency does not involve the same iteration, we insert an offset node to model this fact. In our dataflow IR, we allow the possibility for a node to read the output of a given stream generated by a previous node at a different tick, either past or future. This flexibility in the expressiveness of our dataflow IR needs to be matched by the target language, or be handled in the backend translation phase. If instead, the load/store pair creates a Write After Read (WAR), the corresponding load will be paired with the latest store which modified the value read or will be an input stream nodes with no predecessors. To check this relation across loop iterations we have to consider in the worst-case $O(S * L)$ pairs, where S and L are the sets of all store and load instructions in the current set of sub-graphs, respectively. In the practical cases tested, this procedure has had minimal impact on the overall running time of the tool. At the end of this linking procedure, all the dataflow graphs of the computation should have as nodes with no predecessors only constant values, input streams or memory elements, and the nodes with no successors should be store instructions. This is the most critical phase of the translation as if the tool fails to correctly identify dataflow relations for all subgraphs the translation fails. At the end of this process, only sub-graphs comprised of completely independent computations should remain.

Loop nodes construction

Through the $LoopAnalysisInfo$ provided by LLVM, we are able to identify the loops in the original control flow graph. By traversing the loop info tree we extract relevant information such as the loop trip count and the loop induction variable and identify instructions corresponding to the loop body. This allows us to construct the loop nodes of our dataflow IR, and extract the dataflow sub-graphs belonging to the loop. All the data dependencies across loop boundaries are kept track of through a dedicated interface, while the predecessors of any node in the loop sub-graph are linked to the loop node instead, to obtain a coherent hierarchical graph. Other information such as whether the loop sub-graph contains loop-carried dependencies and all the data dependencies that exist across the loop boundary are collected at this sage. An auxiliary data structure called loop dependency graph is initialized to keep track of the optimizations that will be later applied to each individual loop in the computation. After this procedure is completed, we have successfully transitioned to dataflow IR.

3.4 DFG optimizations

To properly contextualize the target-dependent analysis performed after the dataflow graph construction, the following sections describe the optimization options of our tool. After that, we detail how the target-dependent analysis performed during the translation process can inform the tool on the impact of these optimizations.

```c
#define N 1000

float coeff[] = { ... };

void foo(float a, float* b, float* c) {

    float tmp[N];
    for(int i = 0; i < N; i++){
        tmp[i] = exp(a) + b[i];
        float k = 0;
        for(int j = 0; j < M; j++){
            k += coeff[j] * 0.5;
        }
        c[i] = tmp[i] * k;
    }
    return;
}
```

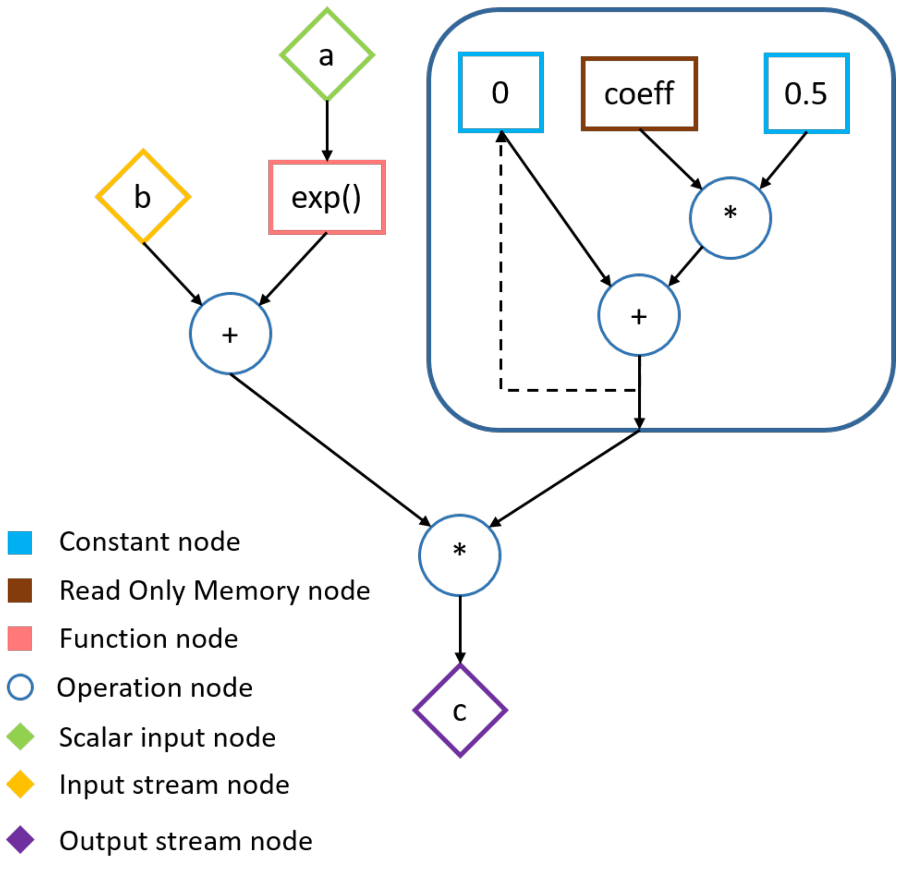


Figure 3.6: Example of a simple function and its DFG IR.

Vectorization

Vectorization changes the data type of the input and output streams of the target function into vector types. If this optimization option is selected, the tool performs an additional transformation pass on the LLVM IR to obtain the vectorized version of the input function. In this transformation pass, given an initial outer loop performing N iterations and vectorization factor v, the iterations of the outer loop are split into two different loops, one inner loop which iterates over the vectorized dimension, and an outer loop that performs N/v iterations. The signature of the function is modified accordingly to the new vectorized types. In this way, we obtain a dataflow IR with one additional nesting level and input and output streams that operate on vector types. This optimization improves the parallelism of the computation by allowing the replication in hardware of the elements of the original dataflow graph and fully utilizing the input and output bandwidth of the target architecture for data transfer. In order for this optimization to be possible, the target architecture must support vector types for input and output stream, as well as a mean to parallelize the new vectorized loop's iterations to achieve higher throughput, as is the case for MaxCompiler. This optimization is particularly useful if the input design is relatively small but iterates over a lot of data. If this is the case, by fully utilizing the FPGA resources we can replicate the hardware and process more data in parallel. The performance gain for this optimization is expected to be a linear increase proportional to the vectorization factor applied and is limited by two main constraints: the amount of hardware resources available on the FPGA and the data transfer bandwidth. A schematic example of the vectorization optimization applied to a simple DFG IR and the resulting hardware replication is shown in figure 3.7.

Loop rerolling

By default, our tool tries to maximize throughput by parallelizing nested loop iterations in the target language as much as possible. This results for example in the replication of hardware functional units and the unrolling of reduction computations. Although this

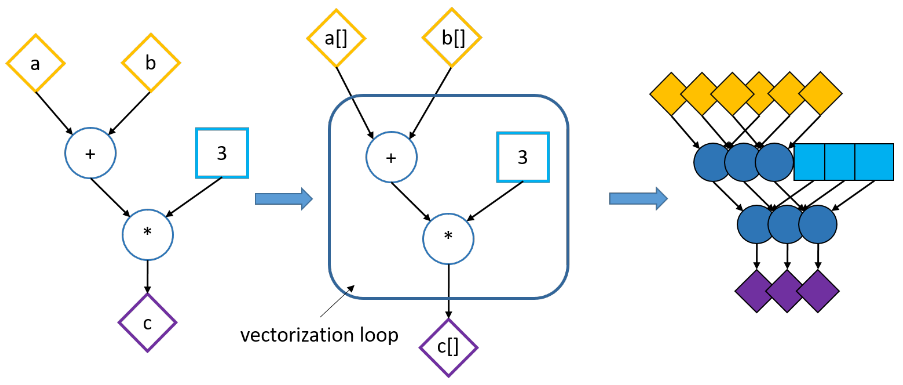


Figure 3.7: Example of a simple DFG IR subject to the vectorization optimization.

strategy can generate effective dataflow designs, in some cases the resources available on the target FPGA do not allow for the maximum amount of hardware replication. Our dataflow IR, through the abstraction of loop nodes, is able to regulate the amount of parallelism applied to nested computations. If a rerolling factor r is given as an optimization option to the tool, the amount of hardware replication involved in parallelizing the computation of loops at the second nesting level in the original function will be divided by r. If for example the original computation contained three loops in total, one loop $L0$ at the first nesting level, and two loops $L1$ and $L2$ at the second nesting level, performing n, m and k iterations respectively, a rerolling factor of r would mean that $\lceil \frac{m}{r} \rceil$ iterations of $L1$ and $\lceil \frac{k}{r} \rceil$ iterations of $L2$ will be executed at each tick of the dataflow kernel. This optimization directly impacts the throughput of our dataflow design, since a valid output value can only be produced after all the nested computations have performed all iterations, which now take r ticks to complete. Therefore, applying this optimization results in a direct loss of throughput proportional to the rerolling factor applied. On the other hand, this optimization reduces the amount of hardware resources used by the design, thus enabling to synthesize functions that would otherwise not fit on the target FPGA. The precise impact of this optimization on hardware resource consumption is detailed in section 3.4.

Cyclic dataflow

There are two fundamental approaches that our tool provides for dealing with nested computations that create a cyclic dataflow graph. By default, nested computations that present loop-carried dependencies are fully unrolled in the target dataflow backend language so that we can produce one useful result at each tick of our dataflow kernel. This approach is very costly in terms of hardware resources, therefore it may not be optimal for some designs. In particular, if we apply this method while applying a rerolling factor r in the case where, for example, a nested loop $L2$ with loop-carried dependencies uses some result computed by a previous nested loop $L1$, the hardware resources dedicated to computing $L2$ would be active once every r ticks since in the remaining $r-1$ ticks $L1$ would be producing intermediate results needed for the completely unrolled computation. An example of how the hardware for $L2$ would behave for a simple sum reduction of 3 elements is shown in figure 3.8. To avoid this waste of resources, we implemented an optimization that allows implementing nested loops with loop carried dependencies as cyclic dataflow computations with feedback arcs. An example of this is shown in figure 3.10. In order to implement this optimization in the target language, we need to be aware of the maximum latency that the nested computation can require to complete one cycle to ensure that the data required by the next cycle has been computed. To achieve this, we implemented an estimation of the latency of our design, detailed in section 3.4. This optimization offers a trade-off between resource consumption and throughput, with a throughput loss directly proportional to the latency of the critical cyclic computation in the dataflow kernel. An example of a simple sum reduction assuming a functional unit with a latency of two cycles is shown in figure 3.10. Since modern dataflow architectures use functional units which are deeply pipelined, the proposed translation approach for this type of cyclic dependencies results in inefficient use of the hardware resources, despite being more efficient than the default strategy. The optimization presented in the next section addresses this issue.

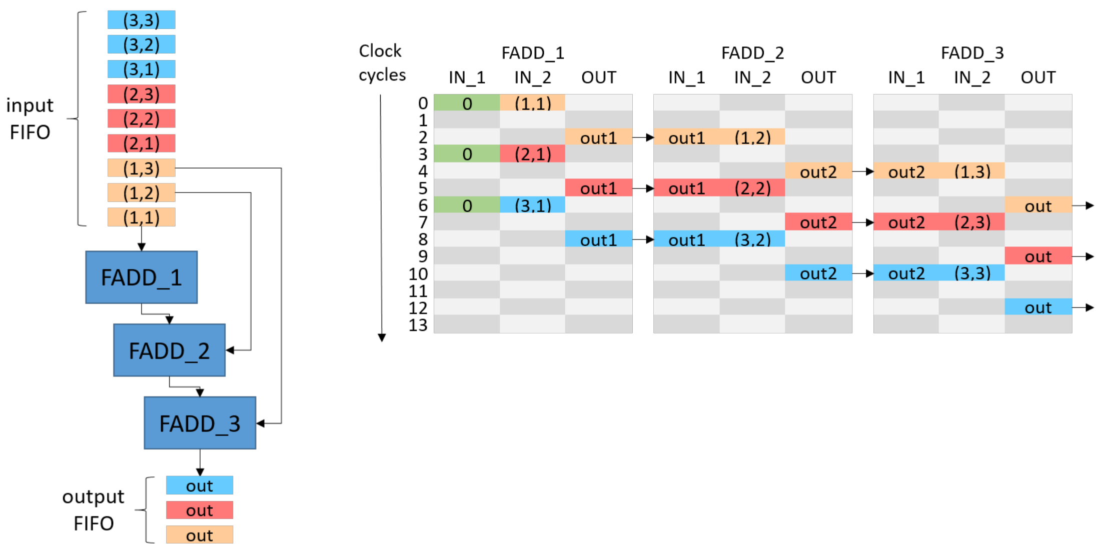


Figure 3.8: Example of fully unrolled sum over 3 elements with a functional unit with latency of 2 cycles. The input stream generates one operand for the sum every cycle, which is directed to the appropriate functional unit. Once every three cycles all the operands are ready and the parallel sum can be completed, resulting in a throughput of one result every three cycles using three functional units.

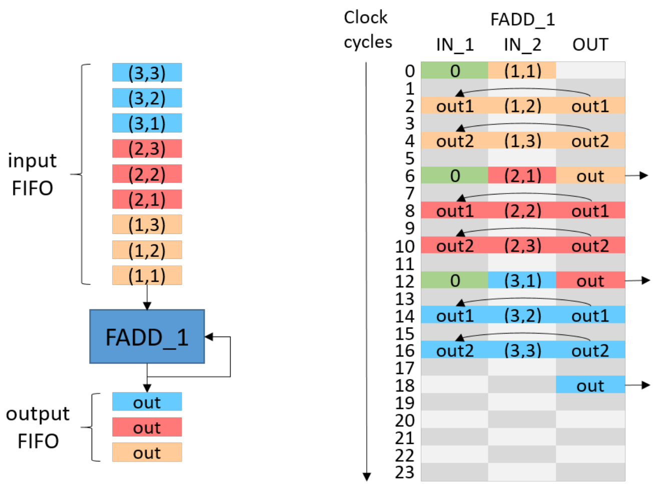
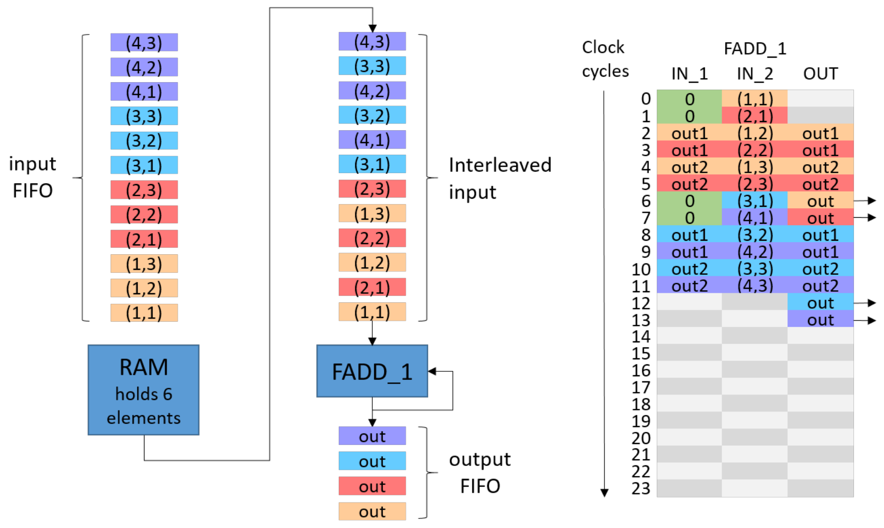


Figure 3.9: Example sum over 3 elements implemented as a cyclic dataflow, with a functional unit with latency of 2 cycles. This results in a throughput of 1 sum every 6 cycles with one functional unit.

Data interleaving

Having observed that in the previous example, given a critical cyclic path with a latency of n ticks we are not using our pipelined functional units in $n-1$ ticks out of n while we could be computing useful data instead, we introduced a further optimization on the computation of cyclic dataflow graphs. During the $n-1$ idle cycles, we can start computing the results for the subsequent $n-1$ input values without waiting for the other iterations of the accumulation to terminate. We have called this optimization cyclic dataflow graph with data interleaving. In order to maintain the correctness of the computation we must ensure that, at each tick, the kernel reads data from the correct input and intermediate results. In practice, this is achieved by using the local on-chip memory. For the first n ticks we store on the local on-chip memory the first n input values and then stop reading new input while we are processing useful data. All the intermediate results relative to the n inputs being processed are also stored in local memory. For every tick, we alternate reading among the n values of input and all intermediate results from the local memories until we complete the computation of the current data and we output the final result. For the next n ticks we store the new input data and initialize the local memories that store intermediate results and we repeat this until all input data has been processed. This optimization allows to mask the latency introduced by cyclic dataflow graphs and obtain a more efficient implementation in terms of resource utilization. An example of a simple sum reduction assuming a functional unit with a latency of 2 cycles with the data interleaving optimization is shown in figure 3.10. It is worth noting that this optimization is only applicable in the absence of loop-carried dependencies on the outer loop of the original function, which ensures that each input data can be considered independently. This means that all the reductions in the design depend only upon inputs read in the same tick. This optimization introduces a possible trade-off between throughput and the use of local on-chip memory. Depending on the latency of the critical cyclic path and the size of the intermediate results that need to be stored at each cycle, we can fully utilize our pipelined functional units at the cost of an



Clock cycles	IN_1	IN_2	OUT
0	0	(1,1)	
1	0	(2,1)	
2	out1	(1,2)	out1
3	out1	(2,2)	out1
4	out2	(1,3)	out2
5	out2	(2,3)	out2
6	0	(3,1)	out
7	0	(4,1)	out
8	out1	(3,2)	out1
9	out1	(4,2)	out1
10	out2	(3,3)	out2
11	out2	(4,3)	out2
12			out
13			out
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			

Figure 3.10: Example sum over 3 elements implemented as a cyclic dataflow with the data interleaving optimization and a functional unit with latency of 2 cycles. This results in a throughput of 1 sum every 3 cycles with one functional unit.

increment in the memory required to implement our design.

Pipelining factor

This optimization option that our tool introduces relates to the calculation of the critical cyclic path. This parameter indicates to the tool which degree of pipelining should be applied for a given functional unit amongst the ones available to the backend synthesis tool. A factor of 1 indicates that the tool will always use the implementation with the most latency available, while a factor of 0 indicates that the tool will always prefer the implementation with the least amount of latency. The trade-off is that choosing an implementation with less latency can reduce the achievable synthesis frequency for the design. Theoretically, it is applicable to any backend synthesis tool which offers different technological implementations of the functional units with different latency to use in the design. In our case, this aligns with MaxCompiler's pipelining factor, which determines the latency of different operations. This has a direct impact on both the throughput in the case of cyclic dataflow graph without the interleaving optimization and on the use of local memory if it is paired with the interleaving optimization since it reduces the interleaving window.

DSP and LUT/FF balance

This optimization option allows to balance the usage of Digital Signal Processors (DSPs), Look-Up Tables (LUTs) and Flip Flops (FFs) by using a different technology mapping for the functional units in the design. Theoretically, it is applicable to any backend that supports equivalent implementations of its functional units with different technology mappings that use a different mix of hardware resources. In our case, this optimization option maps well to the DSP-push target-dependent parameter provided by MaxCompiler.

Memory reshaping and partitioning

To support the use of the FPGA's on-chip memory our dataflow IR uses dedicated memory nodes to model memory elements of different shapes and sizes. In order to ensure that the memory elements modeled in our dataflow IR are optimized for parallel data access and always compatible with the other optimization options of the tool, we have introduced automatic memory reshaping and partitioning. We divided the types of local memory depending on their use within the design based on two main characteristics. First, it is important to differentiate if a memory is accessed through the iteration variable of the outermost loop of the original function. In this case, we reshape the memory so that the corresponding dimension of the memory is the innermost dimension. This is done as a normalization step to isolate this access dimension which will depend on the ticks of the kernel and makes it easier to perform reshaping permutations of the data on the other dimensions. This dimension of the memory is then accessed through a counter that corresponds to the iterations of the outermost loop in the original function. The second relevant characteristic is whether the memory is accessed through an index contained in a nested loop that will be parallelized in the dataflow kernel. In this case, the memory needs to be replicated proportionally to the level of parallelism within the loop, to enable parallel data access. If the loop has been subjected to rerolling, the memory is reshaped by splitting the dimension accessed with the iteration variable of the original nested loop in two, where the innermost of the two contains the elements that are accessed sequentially, while the other contains the elements that are accessed in parallel. The dimension accessed sequentially is always moved to the innermost dimension. If the rerolling factor does not divide the number of iterations of the original nested loop exactly, opportune padding is applied. The implementation of this type of memory is achieved by instantiating several local memories, with depth proportional to the selected rerolling factor, that are accessed in parallel. A different memory element is accessed depending on which iteration of the re-rolled loop is being computed at any given tick. If the nested loop used to access the original data structure is completely unrolled and no access is made through the iteration variable of the outermost loop in the original function, the element is implemented with registers, since it would need to be completely partitioned along all dimensions.

Resource estimation

In order to support an automatic design space exploration, we have implemented a resource estimation analysis which, given a dataflow IR constructed by the translation module, the optimization options selected for the design and a technology library specific to the selected backend, provides an estimation on the amount of resources that the design requires to be synthesized. By traversing the graph of the computation, the resource estimation module counts the number of operations that are present in the design, with a separate counter for each valid opcode or operation type. This counting operation can be performed exactly as a static analysis since our tool requires all the nested loops of the computation to have a constant trip count. The operations that appear outside of any nested computation are counted as one operation each since they won't be subject to any hardware replication in the final design. The operations contained inside nested loops are counted with a multiplier which depends on the original parallelism of the loop, as well as the optimizations that have been applied to it. For example, any operation inside a nested loop performing n iterations subject to a rerolling factor of r, will be counted as $\lceil \frac{n}{r} \rceil$ operations. This methodology allows us to retrieve a precise count of the number of dataflow functional units that will be instantiated in the design. A similar counting operation is done to estimate the use of on-chip memory. Each individual memory node's depth, data width and the number of times the memory will be replicated are counted. To account for the memory used by the data interleaving optimization, we consider the latency of cyclic dataflow graphs estimated through latency analysis as illustrated in section 3.4 as a multiplier for the intermediate values that need to be stored at each cycle. Since this analysis only needs to scan the nodes of the dataflow graph and count them once, it runs in $O(N)$ time, where N is the overall number of nodes in the graph. The analysis produces a report containing all this information that is used together with the target-dependent technology library to estimate the LUT, FF, BRAM and DSP use of the design. The main element that is lacking in the estimation is the resources in the final design dedicated to managing the FIFO queues between the dataflow functional units. Since MaxJ as a target language manages these design elements without requiring the user's intervention and we are still able to formulate very precise estimates before synthesis, we decided to not include it in our analysis. Nonetheless, it is a possible extensions to our current approach that would allow us to improve the precision of the estimates even further.

Latency analysis

To implement cyclic dataflow, we need an upper bound for the critical latency of the cyclic paths in the dataflow graph. Since we know that a cyclic path is only possible in nested computations according to the assumptions imposed on the original input code, we can restrict this analysis only on the sub-graph generated by nested loops. Moreover, during the graph construction phase, each feedback arc is marked so that we can easily verify which nodes of the graph contain a feedback arc, without the need for a cycle detection algorithm. Among all the sub-graphs with feedback arcs at the same nesting level, we need to identify the path with the highest latency and construct a read-enabling structure for all cyclic computations in the design to ensure that the next datum read by the feedback arcs is the correct one at each cycle. To perform this analysis, we have empirically collected the latency observed for all possible functional units in the backend language considering different pipelining factors. Since the overall latency for a complex path was observed to be the sum of the latency of single operations, we can take as a latency estimate the sum of the latency of all the individual operations in the critical cyclic path. It is important for this analysis to be as precise as possible since a higher latency for the cyclic paths in the design directly results in a worse throughput, or a higher memory consumption if the design uses the data interleaving optimization. The latency analysis considers each cyclic sub-graph and identifies the start and end nodes of the possible critical path, corresponding to the pairs of nodes connected through each feedback arc of the sub-graph. Despite the fact that the longest path problem for a general graph is NP-hard [48], we can restrict the case we are considering to that of a directed acyclic graph since we can ignore the feedback arc in the latency computation. Thus we are able to compute the critical latency in $O(N + E)$ time where N is the number of nodes and E is the number of vertices in the sub-graph. First, the nodes of the sub-graph are topologically sorted, then the topologically sorted nodes are scanned once and for each node we update the maximum distance of its successors from the source node based on the distance of the current node and the weight on each arc. Once all the sub-graphs have been processed, we identify the one containing the path with maximum latency, therefore identifying the critical cyclic path in the design, which informs the backend on how to translate the graph into the target language.

3.5 Design space exploration

In order to guide the optimization of the function, our tool generates a performance and resource model specifically tailored for each of the supported optimizations, that considers the target frequency of the design, the latency of the operators, as well as the estimates on memory utilization for implementing Read-Only Memories (ROM) and Random Access

Memories (RAM) required for cyclic dataflow interleaving. Each model uses two sets of variables that can be tuned in order to modify the expected final performance and resource consumption of the implementation. The first is a set of variables v_i for $i \in I$ (or simply $\bar{v}$) related to the optimization to perform, while the second set consists of the variables $\theta_n \in \mathbb{N}$, that specify the technology mapping for a given operator $n \in N$. Each model provides a function $p(\bar{v}, \theta_n)$ that models the throughput of the system (bits / second) and the functions $q_n(\bar{v}, \theta_n)$ that specify the number of instantiations of a dataflow node $n \in N$ in the final system (e.g. number of 32-bits floating-point multipliers, 8-bits adders, ...). The amount of resources r_t of resource type $t \in T$ (e.g.: $T = DSP, BRAM, FF, LUT$) required by the dataflow nodes within the system are estimated as follows:

$$r_t = \sum_{n \in N} c_{n,t,\theta_n} \cdot q(\bar{v}, \theta_n) \tag{3.1}$$

where c_{n,t,θ_n} is the number of resources of type t used by node n under the configuration θ_n. Each type of dataflow node in the design can use a different configuration θ_n. The characterization of the compute nodes given by c_{n,t,θ_n} is performed only once by implementing each node separately as a single kernel function and retrieving the final resource utilization and latency reported by the back-end tool after place-and-route at a given target frequency on the target FPGA, across the possible configurations θ_n. It is worth noting that even if this is a time-consuming task, once the characterization is performed, the achieved results are independent of the application and can be reused. The resource model only takes into account the resources occupied by the kernel and does not consider the resources needed by the communication subsystem. Nevertheless, the resource consumption of the most constrained resource not related to the kernel function is well below 10%. During the design space exploration, the tool takes into account a 15% slack of the total available resources in order to avoid over-constraining the design and leaves enough space for the communication subsystem. In addition to pure dataflow nodes, we also estimate the Block RAM (BRAM) resource requirements needed for im-

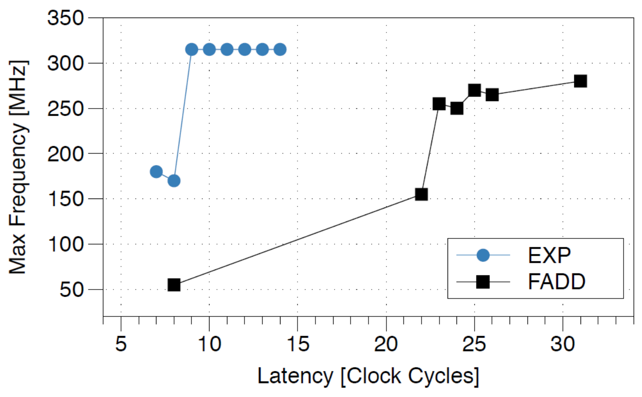


Figure 3.11: Latency and maximum achievable frequency for a single-precision floating-point adder (FADD) and exponential (EXP) operator for different hardware implementations on a MAX4 Galava Card. The maximum frequency values are experimentally derived using micro benchmarks at frequencies in the range $[20, 350]$ *MHz in steps of 5 MHz.*

plementing local ROM and RAM within the design. for a given memory $m \in N$, the function $q_m(\bar{v}; \theta_m)$ specifies the number of partitions in which the memory needs to be divided. The number of BRAM resources required by each memory partition is then estimated as follows:

$$c_{m,BRAM,\theta_n} = \left\lceil \frac{\#BRAM_m}{k} \right\rceil \cdot k \tag{3.2}$$

where $BRAM_m = \left\lceil \frac{size_m}{size_{BRAM}} \right\rceil$ is the minimum number of BRAMs needed to store the data of the memory, while the factor $k = \left\lceil \frac{width_m}{width_{BRAM}} \right\rceil$ takes into account the fact that multiple arrays of BRAMs must be instantiated in parallel to support large memory bitwidths. In the following sections, we provide the expressions of the functions p and q_n of the performance and resource estimation model for the optimizations supported by our approach.

3.5.1 Rerolling model

Our rerolling model requires three different variables:

- $v_0 \in \mathbb{N}^+$: specifies the global rerolling factor to use

- $v_1 \in \{0, 1\}$: equal to 1 if and only if cyclic dataflow is used
- $v_2 \in \{0, 1\}$: equal to 1 if and only if $v_1 = 1$ and if data interleaving is used

When rerolling is applied, the overall throughput of the system is reduced proportionally to the rerolling factor v_0. This is true also for cyclic dataflow with interleaving, but special care must be taken when considering cyclic dataflow without interleaving. Indeed, in this context, the throughput of the implementation also decreases proportionally to the latency of the critical cyclic path. Nevertheless, tools such as MaxCompiler allow exploring different technological implementations for the same operator providing a tradeoff between latency and the maximum achievable frequency. In particular, MaxCompiler exposes the pipeline push optimization that takes a value in the range $[0, 1]$ and can be applied to a specific operator in the code. Figure 3.11 shows the latency and the maximum achievable frequency of different operators implementations synthesized on a MAX4 Galava card. In our model, l_{n,θ_n} and ϕ_{n,θ_n} represent respectively the latency and the maximum frequency of a dataflow node $n \in N$ when implemented using configuration θ_n. Thanks to this characterization, we can easily introduce a bound on the frequency f that the design can achieve depending on the actual dataflow nodes being used within the design and their technology mapping:

$$f \leq \min_{n \in N}\{\phi_{n,\theta_n}\} \tag{3.3}$$

This upper bound is useful when performing the actual design-space exploration but, with designs using a large portion of the FPGA, it is often unfeasible to meet the upper bound due to routing congestion. In addition, we can also compute the latency of the critical cyclic path A as follows:

$$A = \max_{C \in \Gamma}\left\{\sum_{n \in C} l_{n,\theta_n}\right\} \tag{3.4}$$

where Γ is the set of all cycles $C \in \Gamma$ each containing dataflow nodes $n \in C$. Note that, if cyclic dataflow is not applied (i.e.: $v_1 = 0$), $\Gamma = \emptyset$ and $A = 1$ by definition. With these information, we are finally able to express the performance model for the rerolling optimization:

$$p(\bar{v}, \theta_n) = min\left\{ \frac{f \cdot b_{out}}{v_0 \cdot A}, B_{out}, \frac{b_{out}}{b_{in}} \cdot B_{in} \right\} \tag{3.5}$$

where b_{out} and b_{in} are the bitwidth of the input and output streams of the dataflow kernel, while B_{out} and B_{in} are the maximum output and input bandwidth respectively. The number of the dataflow nodes within the rerolled implementation can be computed as:

$$q_n(\bar{v}, \theta_n) = k_0 + \sum_{l \in LR} k_{l,n} \cdot \left\lceil \frac{i_l}{v_0} \right\rceil + \sum_{l \in LU} k_{l,n} \cdot i_l \tag{3.6}$$

LR is the set of nested loops for which rerolling is applied, while LU is the set of nested loops that are fully unrolled. i_l represents the original number of iterations of nested loop l, k_0 is the number of occurrences of node n outside nested loops LU and LR, while $k_{l,n}$ is the number of occurrences of node n within nested loop l. If we apply rerolling without cyclic dataflow ($v_1 = 0$), LR consists of the loops without carried dependencies, while the nested loops with carried dependencies are completely unrolled and belong to LU. However, when we apply rerolling with cyclic dataflow ($v_1 = 1$), then $LU = \emptyset$ and LR consists of all the nested loops that can be rerolled regardless of carried dependencies. Additionally, we also need to estimate the number of partitions for ROM memories within our design. As discussed in section 3.4, we support multi-dimensional ROM in which each dimension $d \in D_m$ can be accessed by constant values ($d \in D_{m,const}$), via a nested loop iteration variable ($d \in D_{m,nest}$), or via the outer loop iteration variable ($d \in D_{m,out}$). We compute the number of partitions required by a ROM m as:

$$q_m(\bar{v}, \theta_m) = \prod_{d \in D_{m,const}} s_d \cdot \prod_{d \in D_{m,nest}} \left\lceil \frac{s_d}{v_0} \right\rceil \tag{3.7}$$

where s_d is the number of elements within dimension d. In this way, we ensure that each constant access can be performed in parallel as well as all the accesses from the iterations of the rerolled nested loops. Finally, to conclude our resource model, we also need to take into account the extra RAMs that are instantiated to perform intermediate storage in case interleaving is applied ($v_2 = 1$). In this case, we consider a memory for each input stream of the dataflow kernel and for each reduction variable resulting from a loop with carried dependencies. These extra memories have a depth of A elements and a bitwidth that depends on the data type of the input stream or reduction variable.

3.5.2 Vectorization model

The vectorization optimization requires a single variable v_0 that represents the vectorization factor. Since vectorization replicates the logic of the kernel in order to perform more iterations in parallel on different input data, the overall performance can be estimated as:

$$p(\bar{v}, \theta_n) = min\left\{v_0 \cdot f \cdot b_{out}, B_{out}, \frac{b_{out}}{b_{in}} \cdot B_{in}\right\} \tag{3.8}$$

where b_{out} and b_{in} are again the bitwidth of the input and output streams of the dataflow kernel, while B_{out} and B_{in} are the maximum output and input bandwidth respectively. We can see how the vectorization factor acts as a multiplier for the resulting throughput until the maximum bandwidth utilization is reached. Regarding resource consumption, the number of dataflow nodes within the resulting design is computed as

$$q_n(\bar{v}, \theta_n) = v_0 \cdot k_0 \tag{3.9}$$

where we simply multiply the original number of nodes in the unoptimized design k_0 by the vectorization factor, since all the hardware of the unoptimized design is completely replicated. Finally, the number of partitions for each ROM within the design is computed as:

$$q_m(\bar{v}, \theta_m) = \prod_{d \in D_{m,const}} s_d \cdot \prod_{d \in D_{m,nest}} s_d \cdot \min\{v_0, s_{out}\} \tag{3.10}$$

where s_d is the number of elements within dimension d of the ROM that are accessed by constant values ($d \in D_{m,const}$), or via a nested loop iteration variable ($d \in D_{m,nest}$).

s_{out} is the number of elements of the dimension accessed via the outer loop variable. If there are no such dimensions, then $s_{out} = 1$ by definition. This model accounts for the fact that if v_0 is greater than s_0, s_0 remains the upper limit on the number of elements that will be accessed in that dimension.

3.5.3 Optimization problem

In order to evaluate the most effective dataflow implementation for a given C function, we employ an automated Design Space Exploration (DSE) to identify the optimal solution. The objective of the DSE is to maximize the overall kernel performance:

$$\underset{\bar{v},\theta_n}{\operatorname{argmax}}\{p(\bar{v},\theta_n)\} \tag{3.11}$$

subject to the FPGA resource constraints:

$$r_t \leq M_t, \forall t \in T \tag{3.12}$$

where M_t is the maximum amount of resource of type T available on the target FPGA, and subject to the frequency upper bound from equation 3.3. The DSE is performed independently on each candidate optimization (vectorization, rerolling, cyclic rerolling, cyclic rerolling with interleaving) and the implementation with the highest expected performance is chosen. The DSE considers increasing values of the vectorization factors until the solution does not fit any more within the device, whereas, for the rerolling optimizations, the process tests increasing rerolling factors until either no more iterations can be rerolled or the design fits within the FPGA. The search for the factors to use for rerolling and vectorization is performed using a binary search approach in order to reduce the number of attempts. Finally, once the optimization variables are set, the selection of the best technology mapping θ_n for $n \in N$ and target frequency f, is performed by solving a Mixed-Integer Linear Programming (MILP) derived by fixing the values of the optimization variables and substituting the vectorization or rerolling model within equation 3.11 and equation 3.12. This approach is effective since most of the non-linearities are removed after fixing the optimization variables and the resulting MILP model is easier to solve. Overall, on our benchmarks, the DSE required less than 10 seconds to complete.

3.6 Backend

In this section, we illustrate how the target-specific backend of our tool is able to generate a synthesis-ready code for MaxCompiler. A Maxeler Dataflow Engine is composed of a dataflow kernel, containing the implementation of the dataflow computation, and a manager, which is responsible for handling the interface between the kernel and the host device. The input of our backend is the dataflow IR composed of one or more independent dataflow graphs and the graph metadata relative to the optimization applied and the latency analysis results. The outputs of our backend are two MaxJ source files, one containing the kernel function and the other containing the code for the manager.

3.6.1 Kernel generation

Since the MaxJ language is a DSL embedded in Java, it relies on proprietary classes and operator overloading to express the dataflow computation. For this reason, a dataflow kernel file needs to import specific libraries and classes, as well as inherit MaxCompiler's Kernel class. Therefore, the kernel code is embedded in a parametric template, an example of which is shown below:

```
package [mypackage];

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
[kernel-specific imports]

class [kernelName]Kernel extends Kernel {

        [kernelName]Kernel(KernelParameters parameters) {
                super(parameters);

            [...kernel code...]
        }
}
```

where the elements enclosed in [ ] are parametric, while the other statements are fixed. These statements import the Kernel class and the DFEVar class, which is the basic data type for dataflow operations in MaxJ. Other import statements are added to the template depending on the backend functions that the kernel code uses. During the instruction generation phase, the template is updated by adding the corresponding imports whenever a node of the dataflow graph is translated using a function or design element which requires a particular import statement. A simple hash table keeps track of the currently imported classes during the translation process.

Instructions generation

To generate the instructions of the kernel, the nodes of the graph are sorted in topological order. After this topological sorting, the nodes are processed one by one, each generating a set of instructions of the final kernel implementation. This is effectively equivalent to traversing the dataflow graph in depth-first order and then generating the instructions by iterating in reverse on the node list. It's important to note that in the case of our backend, differently from a traditional compiler backend, instruction selection and considerations related to the cost of groups of instructions are not a primary concern. This is due to the fact that the source-to-source translation that our tool performs will then be compiled by MaxCompiler, and at this stage MaxCompiler will perform the more fine-grained instruction-level optimizations. Therefore our backend's primary objective is to produce a correct dataflow design that guarantees that the final design implements the optimizations selected by our design space exploration. Since the transition to the LLVM IR dispenses with the original code's variable names, the instruction selection process assigns progressive variable names of the type _<varName>, for example, _a, _b, ...

This has a negative impact on the human readability of the code, but this mainly impacts debugging rather than the end-user of the tool. In fact, the automatically generated code is not meant to be edited by hand but directly synthesized on the target FPGA. For an operation node n which performs the $\oplus$ binary operation on its predecessors $p_1, p_2 \in N$, the statement will be translated as:

$$[VarType]\ [name(n)]\ [assignOp]\ [name(p_1)] \oplus [name(p_2)]; \tag{3.13}$$

where $[VarType]$ corresponds to the type of the result of the operation performed by n, for example $DFEVar$ or $DFEVector$, $name(n)$ is a string returned by the variable name generation function for the node n, $assignOp \in \{=, <==\}$ which correspond to the assignment and connector operators in MaxJ. It is important to note that MaxJ infers the data type of a $DFEVar$ from the type of the $DFEVar$ operators used in its declaration. The type-safety of our translation is inherited by the strongly typed LLVM IR, that our DFG IR is based upon. Therefore, each operation node is characterized by an internal type and all cast operations are performed explicitly. In the case of a simple addition of two $DFEVar$ a and b, the node would be translated as:

```
DFEVar c = a + b;
```

More generally, an operation node can contain different types of operands, as in the case of a selection statement, or have a more complex structure. Moreover, the translation of a node is influenced by the set of active optimizations and technology mapping selection performed in the design space exploration phase. Therefore, a more complete expression of the translation of a node is as follows:

$$S = T_{\oplus}(\bar{v}, \theta_n, name(p_1) \ldots name(p_k), name(n), assignOp_{\oplus,VarType}) \tag{3.14}$$

Where S is a set of statements which represent the translation of the node $n \in N$, $T_{\oplus}()$ is the translation function for the generic operation $\oplus$, that depends on the optimizations selected $\bar{v}$, the technology mapping θ_n, $name(n)$ and $name(p_i)$, where $p_i \in P$, $i \in [1, k]$

are the predecessors of n, and the type of assignment required $assignOp_{\oplus,VarType}$ given the variable type and operation performed.

Loop nodes

The instruction selection process is performed recursively for each sub-graph which represents a nested computation. When a loop node L is encountered during the instruction selection process, the translation S_L generated contains both a prefix and a suffix which enclose the nested computation and the translations $S_1 \ldots S_s$ of all the statements of the loop sub-graph. Depending on whether the autoloop optimization has been selected for the kernel implementation, the loop prefix and suffix can consist of a Java-like for loop, or a series of statements which define the cyclic dataflow control logic and cyclic dependencies. A Java-like for loop is interpreted by MaxCompiler as a macro to replicate the hardware corresponding to the loop statements and fully parallelizes the loop. If the rerolling optimization has been selected, the loop prefix and suffix are modified accordingly, to enforce the desired level of parallelism and ensure that the correct data is read at each cycle.

ROM nodes

To implement ROMs, MaxCompiler needs to be able to initialize the memories with the data before the streaming computation begins. Therefore, in the kernel code, we need to provide the data for each ROM node as a static declaration. In most cases, we can simply create a static declaration in the kernel function. In the case of a mono-dimensional memory $myMem$ containing N 32-bit floating-point data, this would look like:

```
float myMemData[] = { {...}, ... };
...
Memory<DFEVar> myMem =  mem.alloc(dfeFloat(8,24), N);
myMem.setContents(myMemData);
```

However, for ROMs containing a lot of data, the declaration statements can become very large, and after a certain threshold, this is not supported by the compiler. If this is the case, we add to the kernel file the $getArrayFromFile(fileName)$ static function, which reads the values for a given memory node from a file. This file is generated automatically with the appropriate format during the code generation. This method of initialization has no effect on the kernel performance since the initialization code is only executed my MaxCompiler when it synthesizes the kernel. For multi-dimensional memories, the initialization code can become more complex, and some optimizations require the initialization of $ArrayList$ of memories. All this complexity is handled at compile time by automatically generated helper functions which are added to the kernel code.

IO stream nodes

The translation of the input and output stream nodes serves two important purposes: firstly, it declares the data types of the inputs and outputs of the kernel function, in addition, it informs the compiler on how the kernel interfaces with the host code. MaxCompiler allows fixed and floating-point data types, and it supports types of different bitwidth. At present, our tool does not optimize the bitwidth of the data types of the original C function, and it simply translates the standard primitive C types with the corresponding MaxJ types. Namely, signed integers of different bitwidth (e.g. short, int, long) are translated with their MaxJ counterparts, $dfeInt(b)$, with $b \in B = \{8, 16, 32, 64, 128\}$, unsigned integers are translated as $dfeUInt(B)$, and floating-point numbers are translated as $dfeFloat(e, m)$ with $e \in E = \{8, 11\}$ representing the number of bits available for the exponent and $m \in M = \{24, 53\}$ representing the number of bits available for the mantissa. According to the selected optimization options, the data types can be $DFEVector$ types of appropriate length. MaxJ transparently supports interfaces based on vector types, therefore this modification has a minimal impact on the translation process. The IO streams declarations also take a string as an argument, corresponding to the name that the manager will use to refer to that stream when interfacing the kernel and the host-code. In the translation, we name the interfaces with the same variable names generated by the *name*(n) function when translating the corresponding node n. Since *name*(n) never returns a given variable name more than once, our translation is in SSA form, and the interface name is guaranteed to be unique. After the kernel translation process is completed, we generate a file listing the kernel interfaces with the relative data types that will be used by the manager generator.

3.6.2 Manager generation

The manager generation is relatively simple compared to the kernel code generation since it has a very regular structure that uses various MaxCompiler interfaces to regulate the behavior of the dataflow kernel. Similarly to the kernel generation, the manager generation is based on a parametric template:

```
import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2. \
    managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2. \
    managers.engine_interfaces.InterfaceParam;

class [kernelName]Manager extends CustomManager {
    [manager global variables]
    [kernelName]Manager(EngineParameters engineParameters){
        super(engineParameters);
        [manager configuration settings]
        KernelBlock [kernelName]Kernel =
        addKernel(new [kernelName]Kernel(
            makeKernelParameters("[kernelName]")
            ));
        [DFE Links]
    }
    static EngineInterface interfaceDefault(){
        EngineInterface eint = new EngineInterface();
        [interface parameters settings]
        [ticks settings]
        [streams settings]
        return eint;
    }

    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
            [kernelName]Manager manager =
                new [kernelName]Manager(params);
            manager.createSlicInterface(interfaceDefault());
            manager.build();
    }
}
```

within this template, we add all the code that manages the kernel: how it communicates with the host-code, its configuration and target clock frequency and how much data it expects when executing. In the following paragraphs, we detail how each section of the template is filled.

Manager global variables

These are global variable declarations that are generated so that the manager code can refer to important constant values such as the degree of rerolling used when applicable, the latency of the critical cyclic path, as well as the size of the data streams when it happens to be known at compile time.

Manager configuration settings

Specifies some configuration settings used by MaxCompiler during the synthesis process. One of these settings is the target synthesis frequency specified in MHz.

DFE Links

In this section of the code we specify the correspondence between input and output streams declared in the kernel code and elements of the host-side function signature. Here we use the information on IO streams provided by the kernel generation to construct the appropriate interfaces. An example of input $DFELink$ would be:

```
DFELink [streamName] = addStreamFromCPU("[streamName]");
[kernelName]Kernel.getInput("[streamName]") <== [streamName];
```

The declaration of output $DFELink$ uses analogous proprietary functions.

Interface parameters settings

This portion of the code is used to add those parameters of the host-side function that are not used only by the dataflow kernel, but also in the manager code. In our tool, this is primarily used to allow the host-side call to the kernel to pass as a parameter the length of the input streams. This allows using the same kernel design on datasets of different sizes.

Ticks settings

This section of the code specifies how many ticks the kernel needs to perform for a given input dataset. By using information related to the size of the input data, the critical cyclic latency and the rerolling factor, we are able to statically determine how many ticks are required to process all the input.

Streams settings

In this section, we set the size of input and output streams. The size in bytes of each stream is computed statically using the information on the host-side data type of each parameter and the size of the input dataset.

Once we have generated the kernel code and the manager code, we use a bash script to generate a test host code in C which calls the original C function and runs a software simulation for the kernel on the same set of randomly generated data with ranges that can be specified in the input C template. This is intended as a preliminary correctness test, we expect the user to customize the host-side code according to his/her needs and run the hardware synthesis for the kernel.

■

Chapter 4

Experimental evaluation

4.1 Experimental settings

To evaluate the effectiveness of the proposed approach, we have tested our dataflow optimization framework on several applications. The first is an application that consists of a series of filters used in the context of image processing to sharpen images, increasing the contrast between bright and dark regions to bring out features [49]. The second is an Asian Option Pricing algorithm from the domain of finance and the third is a Variational Monte Carlo algorithm used in electronic structure theory. The following sections describe the algorithms and the results obtained by our solution in terms of speedup with respect to a pure software implementation for different parameter configurations and the hardware resources required to implement the designs. Whenever possible, we also compare the results of our implementation with state-of-the-art hand-optimized FPGA implementations of the same algorithm. The results were obtained on a testing system consisting of a host machine with an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz connected via PCI-e gen1 x8 to a MAX4 Galava board. This board contains a Stratix V Intel FPGA. The software baselines are single-thread implementations compiled with gcc 4.4.7 with -O3 level optimizations and executed on the same machine. Finally, Gurobi Optimizer 7.5 [50] has been used as a solver for the MILP models to identify the optimization parameters during the design space exploration phase.

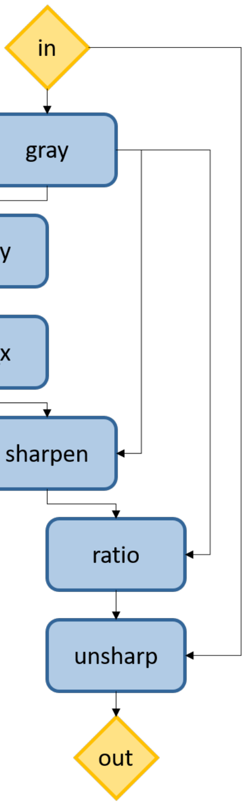


Figure 4.1: The sequence of filters applied by the sharpen filter on the input image. The figure shows the filters and their data dependencies

4.2 Case studies

In this section, we present in detail the experimental results obtained from the case studies selected to evaluate the methodology proposed in this thesis.

4.2.1 Sharpen filter

A schematic representation of the filters applied within the sharpening algorithm and the corresponding kernels is shown in figure 4.1. Our framework is able to generate a synthesizable version of the code from the input C code in a fully automatic way. However, the unoptimized implementation of the algorithm does not harness the potential for parallel computation of the FPGA and can be vectorized until the maximum bandwidth for the target board is reached. In this case, implementing the application without any optimization leads to overall resource consumption of 10% of the most constrained resource and bandwidth utilization of about 26%. In this case, our tool identified a solution using a vectorization factor of 8 which achieves a bandwidth utilization close to the maximum PCI-e gen1 x8 peak bandwidth. Table 4.1 shows different implementations of the sharpening algorithm using different values of the vectorization factor. As can be noted, using a vectorization factor higher than 8 does not bring any benefit due to the data transfer bottleneck and unnecessarily increases resource utilization, while a smaller vectorization factor produces sub-optimal implementations. Overall, we achieved a speedup of 15.85x compared to the CPU-based single thread implementation simply by running our tool on the original code of the application and synthesizing the final system with MaxCompiler. Although the speedup obtained is substantial, it may not appear enough to justify the use of a hardware accelerator, even though the design was obtained automatically from an unoptimized code. On the other hand, this result clearly shows that our methodology is able to find the optimal implementation for the target system: for a vectorization factor of 8 we are able to use all the available communication bandwidth while using only 46% of DSPs, which are the next most used resource in the design. This highlights the fact that the algorithm performs a relatively simple computation while transferring a large amount of data. Since our tool allows to target different FPGAs by specifying the characteristics of the hardware, the user could experiment to test the performance of the optimized designs proposed by our tool on different target architectures. For example, a newer generation of PCI-e could improve performance results. A next optimization step for the user could be the implementation of an image compression strategy to reduce the amount of data transferred and possibly use the resources on the FPGA more efficiently. An optimization of this kind falls outside of the current purpose and capabilities of our tool. We can see from table 4.2 that our resource estimation module is able to predict before synthesis the resources used by the final implementation with a maximum error of 5.76% in the case of BRAMs. As mentioned in chapter 3 in the resource estimation section, this is partially due to the fact that we do not take into account the BRAM employed in the design to handle the FIFO between functional units

Table 4.1: Summary of multiple implementations of the sharpen filter algorithm.

Vector. Factor	Rerol. Factor	Cyclic DFG	Interl.	Freq.	DSP Push	Pipel. Push	Speedup vs CPU	Bandwidth [MByte/s] Input	Output
no	no	no	no	200 MHz	1.0	1.0	4.58	550	550
2	no	no	no	200 MHz	1.0	1.0	8.86	1,064	1,064
4	no	no	no	200 MHz	1.0	1.0	15.37	1,846	1,846
8	no	no	no	200 MHz	1.0	1.0	15.85	1,905	1,905
16	no	no	no	200 MHz	1.0	1.0	15.62	1,877	1,877

Table 4.2: Summary of resource utilization for the multiple implementations of the sharpen filter algorithm shown in table 4.1.

Vector. Factor	Rerol. Factor	Cyclic DFG	Interl.	DSP Push	Pipel. Push	Kernel Resources (estimation error) [%] LUT	FF	BRAM	DSP
no	no	no	no	1.0	1.0	3.63 (-0.18)	2.37 (-0.13)	4.05 (-0.59)	5.86 (0)
2	no	no	no	1.0	1.0	7.03 (-0.14)	4.60 (-0.11)	8.20 (-1.27)	11.72 (0)
4	no	no	no	1.0	1.0	13.9 (-0.12)	9.05 (-0.07)	15.77 (-1.90)	23.44 (0)
8	no	no	no	1.0	1.0	27.52 (+0.04)	17.97 (-0.02)	31.05 (-3.32)	46.88 (0)
16	no	no	no	1.0	1.0	54.96 (+0.16)	35.78 (+0.13)	61.23 (-5.76)	93.75 (0)

in our estimation.

4.2.2 Asian option pricing

The Asian Option Pricing algorithm based on Curran's approximation model [52] is a compute-intensive algorithm used in finance to compute the pricing of Asian options. Due to the nature of these options, the calculation of the final price depends on the prior price of the option across a fixed interval of time. A fixed-point implementation of the algorithm has been proposed in [51]. To calculate the pricing of Asian Options,

Table 4.3: Summary of multiple implementations of the asian option pricing algorithm with 30 averaging points.

Vector. Factor	Rerol. Factor	Cyclic DFG	Interl.	Freq.	DSP Push	Pipel. Push	Speedup vs CPU	Bandwidth [MByte/s] Input	Output
no	5	no	no	220 MHz	0.1	1.0	99.5	1,485.43	165.05
no	4	yes	no	220 MHz	0.1	0.3	13.2	196.42	21.82
no	4	yes	yes	210 MHz	0.1	0.3	118.4	1,767.64	196.40

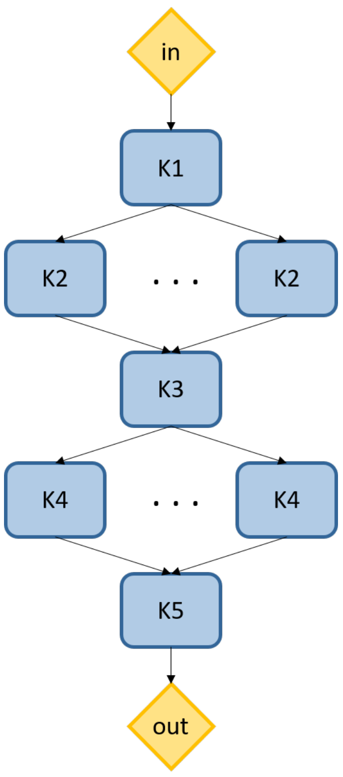


Figure 4.2: The overall structure of the Asian Option Pricing application, as implemented in [51]

Table 4.4: Summary of multiple implementations of the Asian option pricing algorithm with 780 averaging points.

Vector. Factor	**Rerol. Factor**	**Cyclic DFG**	**Interl.**	**Freq.**	**DSP Push**	**Pipel. Push**	**Speedup vs CPU**	**Bandwidth [MByte/s]**	
								Input	**Output**
no	no	no	-	-	-	-	-	-	
no	98	yes	no	215 MHz	0.1	0.3	10.6	7.90	0.88
no	98	yes	yes	215 MHz	0.1	0.3	101.0	75.11	8.35

Table 4.5: Summary of resource utilization for the multiple implementations of the Asian option pricing algorithm with 30 averaging points.

Vector. Factor	**Rerol. Factor**	**Cyclic DFG**	**Interl.**	**DSP Push**	**Pipel. Push**	**Kernel Resources (estimation error) [%]**			
						LUT	**FF**	**BRAM**	**DSP**
no	5	no	no	0.1	1.0	59.76 (-0.70)	36.88 (+1.36)	51.66 (-6.40)	48.05 (+0.39)
no	4	yes	no	0.1	0.3	63.70 (-0.68)	40.17 (+1.26)	52.25 (-5.03)	59.77 (+0.39)
no	4	yes	yes	0.1	0.3	64.12 (-1.10)	40.46 (+0.97)	53.61 (-5.81)	59.77 (+0.39)

Table 4.6: Summary of resource utilization for the multiple implementations of the Asian option pricing algorithm with 780 averaging points.

Vector. Factor	Rerol. Factor	Cyclic DFG	Interl.	DSP Push	Pipel. Push	Kernel Resources (estimation error) [%] LUT	FF	BRAM	DSP
no	no	no	-	-	-	-	-	-	
no	98	yes	no	0.1	0.3	64.20 (-1.18)	40.46 (+0.97)	53.03 (-5.81)	59.77 (+0.39)
no	98	yes	yes	0.1	0.3	64.67 (-1.65)	40.93 (+0.51)	54.39 (-6.59)	59.77 (+0.39)

the algorithm approximates the pricing by considering the market price of the asset at different averaging points across time. This is the reason why most of the computation can be parallelized. Figure 4.2 shows the overall structure of the Asian Option Pricing algorithm. The application is composed of five asynchronous kernels, reported in Figure 3 as K1, K2, K3, K4, and K5. Each kernel performs part of the Curran's algorithm, communicates with the other kernels by means of FIFOs, and leverages fixed-point data types to reduce resource usage while satisfying the accuracy constraint typical of financial applications. Finally, kernel K4 exploits the normal cumulative distribution function (NCDF) to easily compute the Asian put and call options. Maxeler's library provides functionHART to efficiently compute an accurate approximation of the NCDF. functionHART is implemented with a fixed-point piece-wise polynomial approximation generated at hardware compile time using the Remez algorithm [53]. The NCDF function is also the bottleneck in terms of maximum synthesis frequency in case of this design. In fact, the empirical evaluations ran on this NCDF implementation show that it cannot be synthesized at a frequency higher than 220 MHz. This information was integrated with the technological library of the tool. Our methodology is able to produce an automatically optimized implementation of the algorithm with both 30 and 780 averaging points as presented in [51] thanks to the rerolling optimization as well as the optimized translation for loops with loop-carried dependencies. The comparisons with the software baseline for different configurations are shown in tables 4.3 and 4.4. For the version with 30 averaging points, the best configuration uses a rerolling factor of 4, as a lower rerolling factor would require more hardware resources than the ones present on the target FPGA, and uses cyclic dataflow graphs with the interleaving optimization and was synthesized with a frequency of 210 MHz. This version has a speedup of 118.4x with respect to the baseline software implementation. We also compared the DFE execution times reported in [51] of the design optimized by hand. With respect to the version with 30 averaging points, we obtained a speedup of 1.23x. Since the hand-optimized design achieves a lower rerolling factor than our implementation due in part to the fact that it is a fixed-point implementation, we believe that the speedup obtained is caused by the different bandwidth limitations of the two systems. It is worth noting that the implementations that used the cyclic dataflow graph optimization allowed a lower rerolling factor of 4, thanks to their more efficient use of resources. On the other hand, the version with 780 averaging points could not fit on the target board without applying the cyclic dataflow graph optimization. In this case, the best configuration used a rerolling factor of 98 and the interleaving optimization, resulting in an 87x speedup with respect to the software baseline. With respect to [51] our implementation shows a speed down of about 0.5x, in line with the fact that our design has an unrolling factor of 8, while the hand-optimized design computes 15 averaging points in parallel. Nevertheless, we have shown how our framework shortens the gap between the hand-optimized and the automatically-optimized design. The resource utilization estimated for the designs by our resource estimation model are shown in tables 4.5 and 4.6. The maximum error is of 6.59% relative to the BRAM utilization, for similar reasons to the ones mentioned in section 4.2.1.

4.2.3 Quantum Monte Carlo Simulation

Quantum Monte Carlo (QMC) is a blanket term used to denote a set of related methodologies that are used in approximating expectation values to quantum mechanical observables through the (time-independent) Schrödinger equation (TISE). By casting the TISE into integral form, one is able to stochastically sample the many-electron wavefunction in an accurate and scalable way. In particular, two varieties of QMC are favored: Variational Monte Carlo (VMC) and Diffusion Monte Carlo (DMC).

Rather than proceed by giving a high-level overview of the entire application, we instead direct the reader to several excellent reviews [54, 55, 56] of QMC techniques in general, and instead focus upon the parts of the application which are amenable to acceleration through dataflow computing. Furthermore, we direct the reader to a state-of-the-art implementation [57], where the porting to a dataflow platform has been manually undertaken.

The overwhelming computational hotspot for both VMC and DMC applications is the calculation of the so-called trial wavefunction. For systems exceeding a handful of electrons, the computational overhead associated with computing the trial wavefunction exceeds 80% of the total runtime [58]. As such, the trial wavefunction evaluation kernel is a prime candidate to offloading to a hardware accelerator.

The trial wavefunction evaluation kernel can effectively be distilled into the computation of a Slater matrix, $\mathcal{D}$, whose elements are formed from a number of scalar-vector-accumulations,

$$\mathcal{D}_i(\vec{r}) = \sum_{j=1}^{N_{\mathrm{AO}}} \phi_j(\vec{r}) \cdot \mathcal{C}_i \,. \tag{4.1}$$

where $\mathcal{D}_i$ and $\mathcal{C}_i$ correspond to the i^{th} rows of Slater and coefficient matrices, respectively. The N_{AO} atomic orbitals, $\{\phi(\vec{r})\}$, have simple functional form

$$\begin{aligned}\phi_j(\vec{r}) &= \sum_{k=1}^{N_p} d_{jk} \exp(-\zeta_{jk}|\vec{r} - \vec{R}_j|^2) \\ &\equiv \sum_{k=1}^{N_p} d_{jk} \mathcal{N}(\vec{R}_j, (2\zeta_{jk})^{-1}) \,, \end{aligned} \tag{4.2}$$

where $\mathcal{N}(\vec{R}_j, (2\zeta_{jk})^{-1})$ is the gaussian distribution centred on $\vec{R}_j$ with variance $(2\zeta_{jk})^{-1}$. This linear combination of N_p primitives has contraction coefficients d_{jk}. The free variable at which we sample the gaussian, $\vec{r}$, is used to undertake the Monte Carlo. For a standard VMC, the number of independent samples per Monte Carlo step is the product of the number of electrons in the system being studied, n, and the number of concurrent Monte Carlo samplers, N_w, the latter having a magnitude of 10^3 and above.

For the molecular system studied in this work, a box of 64 molecules of the hydrogen dimer in a crystalline geometry, there exist 128 atomic orbitals (each comprising six primitives), 128 electrons and 8192 Monte Carlo samplers. While the system is moderately sized, the complexity of the trial wavefunction has been simplified considerably. The actual computational kernels are, however, representative of ubiquitous functional forms in QMC, and the implementation described in this work is easily extensible to more complex systems. The results of the accelerations of the VMC algorithm with different optimization options are shown in table 4.7. In this case, the bottleneck for the maximum theoretical synthesis frequency is given by the exponential function's implementation that caps the frequency at 280 MHz. However, due to the size of the design and the complexity of the routing, the maximum frequency obtained in practice is 230 MHz. Without the cyclic dataflow graph optimization, the design could not fit on the target board even with the loops completely rerolled. For this design, the best configuration has a rerolling factor of 128 and the interleaving optimization. With this configuration, the design shows a 26x speedup with respect to the software baseline. To compare our solution with the one presented in [57] we compared the ideal bandwidth of the system described in the paper with the real bandwidth achieved by our solution and found a discrepancy of 7% in favor of the ideal bandwidth. It is important to note that the introduction of the interleaving optimization has a noticeable impact on the use of hardware resources, and the DSE takes this into account when it computes the optimal configuration. The resource estimation model was able to estimate resource utilization with a maximum error of 5.76% relative to BRAM utilization. The reason for this result is analogous to the one mentioned in section 4.2.1. A summary of the estimates is shown in table 4.8.

4.3 Results evaluation

The experiments illustrated in this chapter showcase the viability of the proposed methodology as a way to implement FPGA-accelerated dataflow applications. The results show that our automatically optimized designs consistently outperform their software counterparts. More importantly, we were able to show that our designs in some cases

Table 4.7: Summary of multiple implementations of the VMC algorithm.

Vector. Factor	**Rerol. Factor**	**Cyclic DFG**	**Interl.**	**Freq.**	**DSP Push**	**Pipel. Push**	**Speedup vs CPU**	**Bandwidth [MByte/s] Input**	**Output**
no	128	no	no	-	-	-	-		
no	128	yes	no	225 MHz	1.0	0.3	2.7	91.38	89.29
no	128	yes	yes	230 MHz	1.0	1.0	26.2	879.36	859.22

Table 4.8: Summary of resource utilization for the multiple implementations of the VMC algorithm.

Vector. Factor	**Rerol. Factor**	**Cyclic DFG**	**Interl.**	**DSP Push**	**Pipel. Push**	**Kernel Resources (estimation error) [%] LUT**	**FF**	**BRAM**	**DSP**
no	128	no	no	-	-	-	-	-	-
no	128	yes	no	1.0	0.3	29.78 (+1.13)	19.65 (+0.59)	29.20 (+0.49)	72.66 (+0.00)
no	128	yes	yes	1.0	1.0	30.50 (+0.85)	21.80 (+1.10)	59.42 (-5.76)	72.66 (+0.00)

have comparable or better performance than their respective hand-optimized versions. This is ultimately the goal of our methodology since we want to be able to offer the benefits of optimized hardware acceleration at a much lower development cost. In some cases, like for the Asian option pricing algorithm, we are able to see that although for the design with 780 averaging points our methodology yields a 0.5x speed-down, this result is not a methodological limitation: by implementing a variable range estimation procedure, we may be able to introduce an optimization option which uses fixed-point types of different length to further reduce resource consumption within the current framework. With regards to the resource estimation results, we can observe that our models provide very precise estimations in most cases, with the possible exception of the BRAM use estimations, for which the errors remain in a reasonable range below 7%. This confirms that the use of the synchronous dataflow paradigm to express the computation provides a good model for automatic design space exploration before synthesis.

Chapter 5

Conclusions and Future work

In this chapter, we present the final considerations on the proposed methodology for the design of dataflow-based kernels on FPGA. In section 5.1 we outline the contributions of the work presented as well as its limitations. In section 5.2 we describe how the work presented could be expanded and improved.

5.1 Contributions and limits

The work presents a methodology to design optimized dataflow designs for FPGAs that aims at reducing the design time as well as the amount of expertise required to produce an optimized implementation. In this context, we can summarize the main contributions of this work as follows:

- we provide a complete design flow, from an unoptimized C implementation of a software function to a synthesizable optimized dataflow kernel on an FPGA;
- our tool uses Clang, a common C frontend compiler, and does not require the user to modify the input code with FPGA-specific directives or pragmas;
- we propose a code transformation methodology that leverages the LLVM compiler framework to transform an imperative code into a dataflow description of the same computation;
- we propose a custom dataflow intermediate representation on which we are able to

apply various target-agnostic optimizations;

- we implement different target-specific and architecture-aware optimizations targeting MaxCompiler as a backend for the synthesis of our dataflow designs;
- we provide resource estimation and latency estimation analysis to measure the impact of our code optimizations on throughout and resource consumption;
- we implement an automatic design space exploration that selects the optimal set of optimizations to apply for a given input function;
- we provide a backend translation of our dataflow IR into optimized MaxJ code, providing all the elements necessary to initiate the synthesis process on the target FPGA;

Although the proposed methodology shows promising results, it also has some limitations. The ability to translate a software function written in C into a dataflow kernel depends upon several restrictions to the characteristics of the input code. This is partly due to the fact that hardware acceleration generally has less flexibility in its design compared to a general-purpose software implementation, and partly due to some limitations in the code analysis and transformation capabilities of our tool. The transition from software to HDL is a very complex problem that poses numerous challenges even in the restricted case of dataflow computations, and a more sophisticated version of our current frontend optimization process could most likely expand the set of accepted input functions of our tool. Another limitation of our approach is that, although the final result of our transformation and optimization process may vary substantially depending on how the user has expressed the original C function, at present we do not actively provide guidelines for the user to modify his/her code in order to ensure that our tool is able to deliver the best performance.

5.2 Future work

To further develop the presented methodology, we could focus on different aspects of our approach. Firstly, improving the normalization process may result in more flexibility with regards to the restrictions on the input code, so that more applications could be easily accelerated. For example, we could apply loop interchange and loop skewing as well as other loop transformations when appropriate to improve the parallelism extracted from the loops in the input function and manipulate the iteration space to produce the most optimal implementation. In addition, a more accurate estimation of the impact of the FIFOs introduced in the designs on the overall BRAM use may further improve the results of our automatic design space exploration. To achieve this, we could implement an estimation of the FIFO depth required by each interconnection between functional units in the design, depending on the data production and consumption rates of each element and the offsets used in the code to read and write data from streams. For what concerns the optimization options presented by our tool, the addition of a value range analysis and the use of data types with reduced bitwidth and fixed-point numbers could improve substantially the performance of some applications. For example, the state-of-the-art Asian option pricing algorithm used as a case study in the previous section achieved better resource utilization through the use of fixed-point data types. Lastly, to confirm the generality of our methodology we could expand the supported backend synthesis tools, including for example Vivado HLS and SDAccel. This perspective is particularly attractive since it would enable us to test our methodology on high-end FPGA in a cloud environment. For example, by supporting SDAccel as a backend we would be able to deploy our designs on the Amazon F1 instances. Furthermore, given that the proposed methodology aims at providing an accessible way for non-experienced users to program FPGAs, providing our framework as a service in a cloud environment is a very direct and effective way to achieve this goal.

Bibliography

[1] Prasanna Sundararajan. High performance computing using FPGAs. Technical report, Technical Report. Available Online 2010: www. xilinx. com/support . . ., 2010.

[2] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–10, 2014.

[3] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia and Paul Chow. Enabling flexible network fpga clusters in a heterogeneous cloud data center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 237–246, 2017.

[4] Lorenzo Di Tucci, Marco Rabozzi, Luca Stornaiuolo and Marco D Santambrogio. The role of cad frameworks in heterogeneous fpga-based cloud systems. In *2017 IEEE international conference on computer design (ICCD)*, pages 423–426. IEEE, 2017.

[5] David Rohr, Sebastian Kalcher, Matthias Bach, Abdulqadir A Alaqeeliy, Hani M Alzaidy, Dominic Eschweiler, Volker Lindenstruth, Sakhar B Alkhereyfy, Ahmad Alharthiy, Abdulelah Almubaraky et al. An energy-efficient multi-gpu supercomputer. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, pages 42–45. IEEE, 2014.

[6] Derek Chiou. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 124–124. IEEE, 2017.

[7] Grant Martin and Gary Smith. High-level synthesis: past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[8] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015.

[9] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[10] Wesley M Johnston, JR Paul Hanna and Richard J Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

[11] John T Feo, David C Cann and Rodney R Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[12] Neil Hunt. Idf: a graphical data flow programming language for image processing and computer vision. In *1990 IEEE International Conference on Systems, Man, and Cybernetics Conference Proceedings*, pages 351–360. IEEE, 1990.

[13] Nicholas Halbwachs, Paul Caspi, Pascal Raymond and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[14] A Schurr. Bdl-a nondeterministic data flow programming language with backtracking. In *Proceedings. 1997 IEEE Symposium on Visual Languages (Cat. No. 97TB100180)*, pages 394–401. IEEE, 1997.

[15] *LabView*. URL: `https://www.ni.com/en-us/shop/labview.html`.

[16] Jurij Silc, Borut Robic and Theo Ungerer. Asynchrony in parallel computing: from dataflow to multithreading. *Parallel and Distributed Computing Practices*, 1(1):3–30, 1998.

[17] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, 1974.

[18] David E Culler Arvind and DE Culler. Dataflow architectures. *Annual review of computer science*, 1, 1986.

[19] Edward Ashford Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[20] *Vivado HLS*. URL: `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[21] *MaxCompiler*. URL: `https://www.maxeler.com/products/software/maxcompiler/`.

[22] Francesco Peverelli, Marco Rabozzi, Emanuele Del Sozzo and Marco D Santambrogio. Oxigen: a tool for automatic acceleration of c functions into dataflow fpga-based kernels. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 91–98. IEEE, 2018.

[23] Francesco Peverelli, Marco Rabozzi, Salvatore Cardamone, Emanuele Del Sozzo, Alex JW Thom, Marco D Santambrogio and Lorenzo Di Tucci. Automated acceleration of dataflow-oriented c applications on fpga-based systems. In *2019 IEEE 27th annual international symposium on field-programmable custom computing machines (FCCM)*, pages 313–313. IEEE, 2019.

[24] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers and Zhiru Zhang. High-level synthesis for fpgas: from prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[25] Christian Pilato and Fabrizio Ferrandi. Bambu: a modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, 2013.

[26] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36, 2011.

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[28] Michael Delorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E Uribe, F Thomas Jr, Andre DeHon et al. Graphstep: a system architecture for sparse-graph algorithms. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151. IEEE, 2006.

[29] Michael Delorimier, Nachiket Kapre, Nikil Mehta and André Dehon. Spatial hardware implementation for sparse graph algorithms in graphstep. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 6(3):1–20, 2011.

[30] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard et al. Tensorflow: a system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[31] Jeff Bezanson, Stefan Karpinski, Viral B Shah and Alan Edelman. Julia: a fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[32] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron et al. Theano: deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48. Citeseer, 2011.

[33] Jörn W Janneck. Actors and their composition. *Formal Aspects of Computing*, 15(4):349–369, 2003.

[34] Johan Eker and J Janneck. Cal language report: specification of the cal actor language, 2003.

[35] Gul A Agha, Ian A Mason, Scott F Smith and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[36] Tayo Oguntebi and Kunle Olukotun. Graphops: a dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 111–117, 2016.

[37] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon and Rodric Rabbah. Optimus: efficient realization of streaming applications on fpgas. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 41–50, 2008.

[38] William Thies, Michal Karczmarek and Saman Amarasinghe. Streamit: a language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196. Springer, 2002.

[39] Jocelyn Sérot, François Berry and Sameer Ahmed. Caph: a language for implementing stream-processing applications on fpgas. In *Embedded Systems Design with FPGAs*, pages 201–224. Springer, 2013.

[40] Jocelyn Sérot and François Berry. The caph language, ten years after. In *International Conference on Embedded Computer Systems*, pages 336–347. Springer, 2019.

[41] Oliver Reiche, M Akif Özkan, Richard Membarth, Jürgen Teich and Frank Hannig. Generating fpga-based image processing accelerators with hipacc. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1026–1033. IEEE, 2017.

[42] M Akif Özkan, Oliver Reiche, Frank Hannig and Jürgen Teich. Fpga-based accelerator design from a domain-specific language. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2016.

[43] Paul Grigoraş, Xinyu Niu, Jose GF Coutinho, Wayne Luk, Jacob Bower and Oliver Pell. Aspect driven compilation for dataflow designs. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 18–25. IEEE, 2013.

[44] João MP Cardoso, Tiago Carvalho, José GF Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz and Zlatko Petrov. Lara: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 179–190, 2012.

[45] *Perf*. URL: `https://perf.wiki.kernel.org/index.php/Main_Page`.

[46] *Valgrind*. URL: `https://valgrind.org/`.

[47] *Clang*. URL: `https://clang.llvm.org/`.

[48] David Karger, Rajeev Motwani and Gurumurthy DS Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997.

[49] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.

[50] *Gurobi Optimization, Inc. "Gurobi optimizer reference manual,"* 2015. URL: `http://www.gurobi.com`.

[51] Anna Maria Nestorov, Enrico Reggiani, Hristina Palikareva, Pavel Burovskiy, Tobias Becker and Marco D Santambrogio. A scalable dataflow implementation of curran's approximation algorithm. In *2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 150–157. IEEE, 2017.

[52] Alexander Novikov, Scott Alexander, Nino Kordzakhia and Timothy Ling. Pricing of asian-type and basket options via upper and lower bounds. *arXiv preprint arXiv:1612.08767*, 2016.

[53] Jean-Michel Muller and Jean-Michael Muller. *Elementary functions*. Springer, 2006.

[54] WMC Foulkes, L Mitas, RJ Needs and G Rajagopal. Quantum monte carlo simulations of solids. *Reviews of Modern Physics*, 73(1):33, 2001.

[55] Brian L Hammond, William A Lester and Peter James Reynolds. *Monte Carlo methods in ab initio quantum chemistry*, volume 1. World Scientific, 1994.

[56] M Peter Nightingale and Cyrus J Umrigar. *Quantum Monte Carlo methods in physics and chemistry*, number 525. Springer Science & Business Media, 1998.

[57] Salvatore Cardamone, Jonathan R Kimmitt, Hugh GA Burton and Alex JW Thom. Field-programmable gate arrays and quantum monte carlo: power efficient coprocessing for scalable high-performance computing. *arXiv preprint arXiv:1808.02402*, 2018.

[58] Kenneth P Esler, Jeongnim Kim, David M Ceperley and Luke Shulenburger. Accelerating quantum monte carlo simulations of real materials on gpu clusters. *Computing in Science & Engineering*, 14(1):40–51, 2012.

[59] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik and Andrew Wallace. Ripl: a parallel image processing language for fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(1):1–24, 2018.