

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Restructuring Software Code for High-Level Synthesis Using a Graph-based Approach Targeting FPGAs

Afonso Soares Canas Ferreira

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João M.P. Cardoso

July 27, 2018

Abstract

Field Programmable Gate Arrays (FPGAs) are becoming a popular solution for accelerating the execution of software applications. The use of high level synthesis (HLS) tools intends to provide levels of abstraction comfortable to software developers when targeting FPGA-based hardware accelerators. However, the need to restructure the software code and to use adequate directives require both mastering the HLS tool used and FPGA hardware. This dissertation presents a new approach for code restructuring intended to help software developers in achieving efficient hardware implementations. The approach uses an unfolded graph representation, which is generated from program execution traces, together with graph-based optimizations such as folding to generate suitable C code to input to HLS tools, such as Vivado HLS. The experiments show that the approach is capable of generating efficient hardware implementations only otherwise achievable using manual restructuring of the input software code and manual insertion of adequate directives.

Resumo

Field Programmable Gate Arrays FPGAs apresentam se cada vez mais como uma solução para acelerar aplicações de *software*. O uso de ferramentas síntese de alto nível permite níveis altos de abstração acessíveis a programadores de *software*, quando este pretendem utilizar aceleradores em hardware baseados em FPGAs. Mas a necessidade de reestruturar código de *software* e o uso adequado de diretivas requer tanto conhecimento da ferramenta de HLS tal como do *hardware*. Esta dissertação pretende mostrar os nossa investigação em novos métodos para obter reestruturações de código para ajudar a programadores de *software* a obterem implementações em *hardware* eficientes. O nosso método baseia-se num uso de uma representação em grafo do traço de execução de um programa, aliado a otimizações de grafos como enrolamento para gerar código de C adequado a ferramentas de HLS como Vivado HLS. As nossas experiencias demonstram que o nosso método é capaz de gerar implementações mais eficientes, que só seriam possíveis a partir de reestruturação manual do código e uso adequado de diretivas.

Acknowledgments

I would like to thank my supervisor João M.P. Cardoso for the guidance during the work on this dissertation. I also want to acknowledge my colleagues in the Special Computing Systems, languages and tools (SPeCS) lab for all the help given during the development of this dissertation.

I would like to acknowledge INESC TEC for providing a grant during the course of the development of this thesis. The grant was under the project "TEC4Growth -TL -SMILES-5 Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact" funded by the European Regional development Fund.

Afonso Soares Canas Fereira

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Motivation | 1 |
| 1.2 | Objectives | 2 |
| 1.2.1 | Problem description | 2 |
| 1.2.2 | Proposed solution | 3 |
| 1.3 | Organization of the dissertation | 4 |
| 2 | Related Work | 7 |
| 2.1 | State of the art of <i>HLS</i> tools | 7 |
| 2.2 | HLS tools from industry | 9 |
| 2.3 | Overview of HLS tools | 10 |
| 2.4 | Code restructuring | 11 |
| 2.5 | Graph restructuring | 16 |
| 2.6 | Summary | 18 |
| 3 | Description of the Approach | 19 |
| 3.1 | Overview of the frontend | 19 |
| 3.2 | DFG at the frontend | 19 |
| 3.3 | Operation descriptions | 20 |
| 3.3.1 | Graph generation | 22 |
| 3.3.2 | Information lost through tracing | 28 |
| 3.3.3 | Fronted limitations | 29 |
| 3.4 | Overview of backend | 30 |
| 3.4.1 | Backend graph description | 30 |
| 3.4.2 | Structure of the backend | 31 |
| 3.4.3 | First Stage: Graph Initializations | 32 |
| 3.4.4 | Second Stage: Output Analysis | 32 |
| 3.4.5 | Third Stage: Parallel Matching | 32 |
| 3.4.6 | Fourth Stage: Sequential Matching | 33 |
| 3.4.7 | Fifth Stage: Dataflow Optimizations | 33 |
| 3.4.8 | Sixth Stage: Graph Unfolding | 34 |
| 3.4.9 | Seventh Stage: C code generation | 34 |
| 3.5 | Summary | 34 |
| 4 | Backend C Code Restructuring | 37 |
| 4.1 | Initializations | 37 |
| 4.2 | Output Analysis | 39 |
| 4.3 | Parallel Matching | 46 |

| | | |
|----------|--|-----------|
| 4.4 | Sequential matching | 51 |
| 4.5 | Graph Unfolding | 55 |
| 4.6 | Dataflow optimizations | 58 |
| 4.7 | C code generation | 63 |
| 4.8 | Limitations | 66 |
| 4.9 | Summary | 66 |
| 5 | Experimental Results | 69 |
| 5.1 | Experimental setup | 69 |
| 5.2 | Results | 71 |
| 5.3 | Execution time and scalability | 81 |
| 5.4 | Summary | 83 |
| 6 | Conclusion | 85 |
| 6.1 | Future work | 85 |
| | References | 87 |
| A | Benchmark C code | 91 |
| B | Operation Mode | 95 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Representation of the the framework of the approach | 4 |
| 3.1 | Architecture of the frontend | 19 |
| 3.2 | Graph corresponding to statements $a = b$ and $d = b + c$ | 20 |
| 3.3 | DFG of the dot product with vectors of size 4 | 23 |
| 3.4 | Architecture of the backend | 30 |
| 4.1 | DFG of the <i>filter subband</i> benchmark considering an execution with Nz, Ns, Nm and Ny equal to 4, 2, 1024, and 2, respectively | 38 |
| 4.2 | <i>filter subband</i> after stage 1 initializations | 39 |
| 4.3 | <i>Filter subband</i> benchmark after second stage of the tool and considering an execution with Nz, Ns, Nm and Ny equal to 4, 2, 1024, and 2, respectively | 42 |
| 4.4 | Separate sequences for <i>filter subband</i> considering an execution with Nz, Ns, Nm and Ny equal to 8, 4, 1024, and 4, respectively | 47 |
| 4.5 | <i>Filter subband</i> after second stage of the tool and considering an execution with Nz, Ns, Nm and Ny equal to 8, 4, 1024, and 4, respectively | 48 |
| 4.6 | Inner loops of the <i>2D Convolution</i> benchmark during stage 3 four considering an execution with N, K, equal to 2 and 1, respectively | 50 |
| 4.7 | <i>Filter subband</i> after fourth stage of the tool and considering an execution with Nz, Ns, Nm and Ny equal to 8, 4, 1024, and 4, respectively | 54 |
| 4.8 | Unrolled dataflow of pipelined <i>filter subband</i> benchmark of Figure 4.7 by a factor of four considering an execution with Nz, Ns, Nm and Ny equal to 8, 4, 1024, and 4, respectively | 57 |
| 4.9 | Partial sum accumulation applied to unfolded dataflow in Figure 4.8c | 60 |
| 4.10 | Result of memory accesses optimization for the benchmark <i>1D fir</i> | 61 |
| 5.1 | Speedups for <i>filter subband</i> | 72 |
| 5.2 | Speedups for <i>dotproduct</i> | 74 |
| 5.3 | Speedups for <i>Autocorrelation</i> | 75 |
| 5.4 | Speedups for <i>1D fir</i> | 77 |
| 5.5 | Speedups for <i>2D Convolution</i> | 78 |
| 5.6 | Speedups for <i>SVM</i> | 80 |
| 5.7 | Backend execution time in seconds for multiple input image sizes | 82 |

List of Tables

| | | |
|------|---|----|
| 2.1 | Overview of some of the most relevant HLS tools | 10 |
| 2.2 | Overview of some of the most relevant HLS tools | 11 |
| 3.1 | Overview of frontend node information | 21 |
| 3.2 | Overview of backend node information | 31 |
| 5.1 | Benchmark information | 70 |
| 5.2 | Main Frontend DFG information for each benchmark | 70 |
| 5.3 | Description of framework levels | 71 |
| 5.4 | Description of the different versions of input code used to compare results | 72 |
| 5.5 | HLS results for <i>filter subband</i> | 73 |
| 5.6 | HLS results for <i>dotproduct</i> | 74 |
| 5.7 | HLS results for <i>Autocorrelation</i> | 75 |
| 5.8 | HLS results for <i>1D fir</i> | 76 |
| 5.9 | HLS results for <i>2D Convolution</i> | 78 |
| 5.10 | HLS results for <i>SVM</i> kernel | 79 |
| 5.11 | Execution time of backend in ms for each optimization level | 82 |

Listings

| | | |
|-----|--|----|
| 3.1 | Basic graph described in the DOT language | 21 |
| 3.2 | Code of the <i>dotproduct</i> benchmark | 24 |
| 3.3 | Code of dot product with instrumentation code | 25 |
| 3.4 | Dot description of the <i>dotproduct</i> DFG with vectors of size 4 | 26 |
| 3.5 | Examples of kernels | 28 |
| 4.1 | <i>filter subband</i> original source code | 37 |
| 4.2 | <i>Filter subband</i> equivalent output code after stage 3 folding | 49 |
| 4.3 | <i>2D Convolution</i> stage 3 output | 50 |
| 4.4 | <i>filter subband</i> code output after the pipeline of the fourth stage of the tool | 55 |
| 4.5 | <i>ID fir</i> output code after meory access optimizations in Stage 5 | 62 |
| A.1 | <i>filter subband</i> original source code | 91 |
| A.2 | <i>dotproduct</i> original source code | 91 |
| A.3 | <i>Autocorrelation</i> original source code | 91 |
| A.4 | <i>ID fir</i> original source code | 92 |
| A.5 | <i>SVM</i> original source code | 93 |
| B.1 | JSON configuration file example for the <i>filter subband</i> benchmark | 95 |

Acronyms

| | |
|-------|-------------------------------------|
| BRAM | Block Random Access Memory |
| CDFG | Control Data-Flow Graph |
| CPU | Central Processing Unit |
| DFE | Dataflow Engine |
| DFG | Dataflow Graph |
| DSP | Digital Signal Processing |
| FF | Flip Flop |
| FPGA | Field Programmable Gate Array |
| HLS | High Level Synthesis |
| HDL | Hardware Description Language |
| II | Iteration Interval |
| ILP | Instruction Level Parallelism |
| LUT | LookUp Table |
| RAM | Random Access Memory |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuit |

Chapter 1

Introduction

1.1 Motivation

Multiple aspects of our daily lives are only possible due to computers. One of the most important aspect of these machines is their capacity for doing a massive numbers of calculations. There have been many important milestones in the improvement of the computation capacity of computers, such as smaller transistors or multi core processors. A common method of achieving higher performance is utilizing graphical processing units (GPUs) to accelerate application execution [1]. An alternative to this approach is the use of accelerators implemented in field-programmable gate arrays (FPGAs) [2] [3] [4]. FPGAs are a form of reconfigurable integrated circuits. A single FPGA can be reconfigured for many different applications. An FPGA is configured by synthesizing a description of the targeted problem. This mutability aspect enables the designer to accelerate multiple applications without custom building new hardware. When optimized, these FPGA-based accelerators are capable of outperforming GPU based accelerators [5]. Additionally to the high performance, FPGAs are energy efficient.

A custom hardware implementation of an application differs fundamentally from a software one. In the former the hardware is always a CPU and the programmer designs solutions by specifying a sequence of instructions that the hardware must complete in sequential order. There are many different types of CPUs that can be optimized differently, but the fundamental form of programming one is the same. Therefore, programming languages have been designed to describe problems in ways that are clear for programmers and implementable in a CPU in efficient manners. However, in digital hardware design, instead of determining instructions that are executed in order, the designer creates and connects functional units, such as adders and multiplexers, so that the circuit implements the behavior of the given algorithm. This gives the designer a lot of freedom to tailor the hardware to execute faster than a software-based approach. For example, the designer is capable of executing multiple independent operations concurrently as long as the hardware provides enough resources, thereby improving execution of algorithms that have high degrees of instruction level parallelism (ILP). In fact, the circuit can execute entire tasks concurrently if they are independent. However, to actually put this into practice, it is necessary to

describe the algorithm in the form of the flow of the signals through the functional units of the circuit instead of sequenced instructions. Certain tools are capable of programming into FPGAs hardware register-transfer level (RTL) [6] descriptions in hardware-description languages (HDL) such as Verilog [7]. These are capable of expressing the data and control flow of the circuit and take into account factors like clock signals and concurrent operations. However, in doing so they differ drastically from software languages. So, to design optimal hardware one must have many different skillsets and understand very distinct languages than a standard software programmer. Additionally, to describe an entire application at such low level is very time-consuming. These differences create a huge entry barrier to use FPGAs as accelerators.

To solve this there have been developments in the field of high-level synthesis (HLS) which allow designers to describe hardware solutions at higher levels of abstractions, such as a software programming language like C. The intention of employing these higher levels of abstraction is to allow developers to design for hardware with more ease and be able to handle more complex applications without the time consumption of designing every hardware module at low level. The typical HLS flow [8] synthesizes higher levels of abstraction first by scheduling operations to specific clock cycles. Resources are allocated based on the scheduling and then bound to specific hardware units. The HLS tool can then create an implementable HDL description of the input.

The HDL description describes the dataflow and control flow of the hardware implementation of the application. There are many different allocation, scheduling and binding algorithms as well as approaches to defining the control flow of the implementation. Typically, this design flow does not generate implementations as good as manually designed ones, but it allows a higher level of abstraction. However, even though these tools raise the level of abstraction, they still require expertise in hardware to implement optimized solutions in any chosen tool. These tools may accept programming languages like C but the structure of the code has a large impact on the results [9]. Additionally, some tool may require added directives or configurations to generate optimal implementations. So, current HLS tools still have a barrier of entry for programmers. By lowering this barrier, more designers are enabled to use the power of FPGA-based heterogeneous implementations to accelerate code execution, leading to improvements in computing performance and energy consumption. Advancing the state of the art of HLS by increasing its accessibility is the main motivation for this dissertation work.

1.2 Objectives

1.2.1 Problem description

C based languages are a common input for many HLS tools [5]. However, C has many limitations as a language to describe hardware implementations [10]. Thus, HLS tools compensate these limitations by allowing the programmer to guide the synthesis through configurations or directives. Additionally, the structure of the code has a large impact on performance. For example, a multiplication with a variable that actually is constant can be implemented by a shift. Shift registers with

one operand as a constant are far more efficient and easy to implement in hardware than multiplications but for an unexperienced programmer that is not an obvious change. But actually, without changing the multiplication for a shift in the code the HLS tools might instantiate a multiplier that is slower and more resource intensive. This is a simple example and there are much more complex optimizations, such as structuring a loop to be optimally pipelined. A synthesized implementation in hardware is primarily measured in three aspects: The clock frequency, the latency and the resource usage. Optimized C code in HLS can obtain far better results in these aspects than unoptimized C code. Thus, to make C-based HLS more accessible, there needs to be a way to easily restructure the code.

1.2.2 Proposed solution

This dissertation proposes a framework to automatically generate optimized C code. It consists of a frontend that generates a dataflow graph (DFG) from the execution trace. This DFG is then processed in a backend to generate the C output which is input to a HLS tool. This graph based approach is chosen because DFGs are good representations of the dataflow and express very well properties such as parallelism which are essential for hardware implementations. Additionally with a flexible front end it would be possible to generate DFGs from multiple different input languages. This would further increase the accessibility by allowing programmers of different languages to use C-based HLS tools.

This dissertation focuses on the implementation of the back end and the way it manipulates and analyzes the graphs to automatically generate the output C code. It also details the type of graphs the front end should generate, as well as methods to generate them from execution traces.

In this dissertation Vivado HLS [11] is used as the C-based HLS tool, and thus some optimizations, such as directives, take this target into account. Vivado HLS handles all stages of HLS design. It requires timing constraints to schedule the application. Vivado HLS can synthesize loops and implement them using pipeline schemes. It also handles the generation of the memory and interfaces for the applications. The tool also allows the setting of resource constraints either by directly limiting specific resources or by limiting the amount of concurrent operations. All these elements can be manipulated through directives. However, the backend could be reconfigured to target different tools.

Figure 1.1 shows a representation of the framework. The framework implements code restructuring automatically, thus the user only needs to input the source and a few simple configurations such as the inputs and outputs of functions. The quality of an FPGA implementation is measured by the speed and resource usage. Speed is determined based on the clock frequency as well as the total number of clock cycles (latency) of the implementation. FPGAs typically specify 4 main types of resources. Lookup tables (LUTs), which implement the logic, Flip Flops (FF), which are used to store values between clock cycles, DSP units, which are specific units utilized to efficiently implement operations typical for digital signal processing like multiplications and accumulation operations, and BRAMs (Block RAM) which is the type of RAM memory used in the FPGA to store data as memory. Typically, BRAMs only have two read and write ports, so Vivado HLS

needs to partition the memory into multiple blocks to increase the amount of concurrent reads. DSP units could be implemented in LUTs but they would require more area and would be slower.

This dissertation intends to demonstrate that a graph based approach can generate optimized C with improved speed and resource usage while maintaining accessibility for software developers.

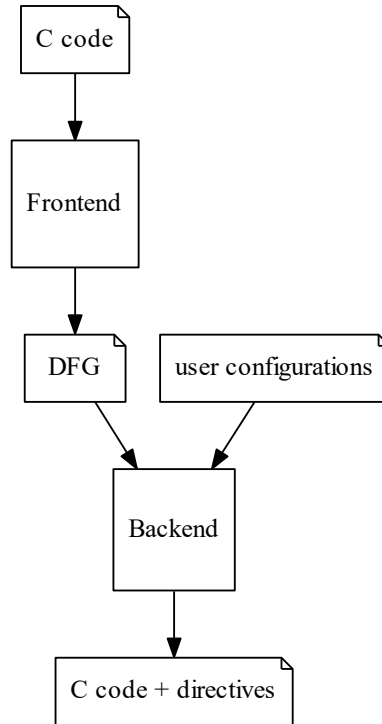


Figure 1.1: Representation of the the framework of the approach

1.3 Organization of the dissertation

The rest of the dissertation is organized in five additional chapters. Chapter 2 presents the related work to the dissertation. This chapter overviews the state of the art of HLS by analyzing different tools. The Chapter 2 also does an in-depth study of the works that deal with optimized C code for hardware. We analyze multiple examples of manually restructured code for different applications and some specific Vivado HLS implementations. Additionally multiple works related to graph analysis and manipulation are presented. Chapter 3 gives an overview of the framework. It starts by detailing the frontend that generates the DFG. Then the chapter presents the structure and implementation of the backend. Chapter 4 describes in depth all of the steps taken by the backend, with multiple examples of graphs and code. Chapter 5 presents the result of the framework on

a set of benchmarks. The final chapter ends the dissertation with some concluding remarks and proposals for future work.

Chapter 2

Related Work

This chapter compiles and details a number of work efforts related to the main topics of this dissertation.

2.1 State of the art of *HLS* tools

A contemporary survey of HLS tools, presented in [5], gives an overview of the landscape of state of the art of HLS tools. This survey introduces multiple tools, both commercial and academic, and indicates some of the common approaches they take to achieve high performance. The survey also features an analysis of four different tools. The results show that academic tools are in parity with commercial tools and that they are capable of achieving higher performance than software-based implementations.

This section details some of the most relevant related approaches. One of the tools is the LegUp HLS tool [12] [13]. The objective of LegUp is to compile standard C code to be then implemented in a hybrid FPGA-based software/hardware system-on-chip. The approach taken by LegUp is to first compile the C code using the low level virtual machine (LLVM)[14] compiler. The compiler optimizes the code through multiple compilation passes, and the LegUp framework modifies these passes to implement HLS optimization algorithms. The compiled instructions are simple enough to be implemented in hardware. The input code is executed on a MIPS processor with profiling capability. The objective of the profiler is to determine which functions to be implemented in hardware. It analyzes and displays the results for each function and using this profile data the user can choose which functions to implement in hardware. The selected functions are then passed through the stages of HLS and implemented in an FPGA, while the remainder run in the MIPS processor. The experiments and evaluations of the LegUp framework indicate significant increases in performance even in comparison to commercial tools.

Another approach is the one taken by the Cameron project [15]. The framework is tailored for image processing algorithms. The framework considers as input language SA-C (Single Assignment C), which is based on the programming language C. The presented framework processes the SA-C code and generates a specific data flow graph (DFG). This DFG is then translated into

VHSIC Hardware Description Language (VHDL) [16] for hardware implementations. The SA-C language is structured so that the compiler can generate an optimized DFG to be implemented in hardware. The framework contains many optimization passes and algorithms, and techniques to port the DFG to VHDL.

A third approach is the language for aggressive loop pipelining (LALP) [17]. This domain specific language is aimed at improving the implementation of loops on FPGAs by maximizing the throughput. Many tools implement loops by statically scheduling the operation and controlling the execution through a finite state machine (FSM). The loop implementation of this approach discards a global FSM and instead operations are executed at specific clock cycles after the start of an iteration. The language allows the designer to specify exactly on what cycle an operation executes, how many pipeline stages some operations have and the number of cycles until the new iteration starts. The execution of operations is controlled by passing along the datapath of the loop, the iteration number and a step signal. An operation is executed when it receives this step signal, thereby imposing a correct execution order without a global FSM. This approach gives the designer a large freedom to optimize the loop, without adding extra complexity. The tool presented in [17] still has some limitations. It requires that designers explicitly deal with data dependencies, otherwise it might schedule the execution of non-independent operations to the same execution cycle. The language also does not support if-then statements, but allows assignments whose values have two distinct sources. The LALP code is implemented in an FPGA by first being translated to a CDFG. This CDFG is then scheduled and balanced. Afterwards, the CDFG is translated into RTL VHDL to be implemented in an FPGA.

A fourth approach is ASC [18]. This approach also uses a new language also called ASC. The language is based on C++ and a specific C library. In this language the designer describes the algorithm as a stream of operations on an input that generate an output. The representation is then implemented on FPGA by having FIFOs at the inputs and outputs and a datapath in between. The language allows many ways for the designer to define variables and memory usage. It also allows users to indicate what aspect of the implementation to optimize. The user can choose between optimizing latency, area or throughput and different parts of the stream can be optimized differently. The tool translates the stream into a graph and operations are scheduled statically and a control block is generated. The hardware modules implemented on FPGA are generated using the PAM-Blox II framework [19], and different modules are chosen depending on the optimizations and implementations of the designer. The Pam-Blox II contains a module-generation library in C++, which generates specific modules based on given parameters. This software-based approach allows the library to be designed in such a way that the tool can be used with many different kinds of FPGAs by, for example, overloading functions.

Another HLS tool is *bambu* [20]. It can support various features of C such as function calls and pointers. The tool generates the VHDL description of the input algorithm and a testbench to verify the description. It can also generate interfaces with software if requested by the user. Additionally, it allows users to integrate commercial tools to implement the generated HDL descriptions in hardware. The tool also applies custom HLS scripts based on these integrated tools.

The user influences compilation and optimization through a configuration defined in a XML file. The design flow starts with a GCC compiler at the frontend that generates a call graph of the input compiled by GCC. The GCC front end allows programmers with knowledge of C to use some GCC optimizations. The next stage of the design flow analyzes the call graph and deals with memory allocation. The updated graph then advances to the next step, which generates the data path and control flow. This generation is influenced by the aforementioned XML configuration file. Bambu instantiates a module for every function and thereby allows the user to uniquely configure the generation of different segments, such as loops or functions, as well as the whole program. The next stage generates the netlist for the data and control paths and, if necessary, the interface for HW/SW integration.

Another approach is presented in [21]. The authors propose a solution based on the dataflow language RCV-CAL. They give a basic explanation of dataflow programming. It consists of a connection of actors that have their own set of actions and only communicate through buffers and tokens. The design flow starts with a behavioral description in RCV-CAL as well as constraints for the implementation and the target architecture. These are fed into a compiler which verifies the description and analyzes it. Afterwards, the hardware or software code is generated, and the hardware code is synthesized for the architecture. The results can be implemented in a FPGA. The code is also lead to a stage that assesses the performance and profiles the result based on the constraints. The results of the profiling are fed back to the compiler to implement necessary changes. Orcc is an open source RCV-CAL compiler. It can translate the input into source code or an IR-representation. Their approach uses Xronos tool which uses Orcc as a back end to generate an IR-representation. It creates a CDFG and sends it to a tool called Openforge to generate a Verilog representation. The authors present the approach based on a Xronos implementation specifically targeting Vivado HLS. They continue to use Orcc as a back end but now use Xronos to generate optimized C code that is fed to the Vivado HLS tool. The authors explain how they implement actor communication through an AXI Stream interface, action selection as well as some optimizations such as expression splitting to improve resource usage of Vivado HLS.

2.2 HLS tools from industry

There are several HLS tools from Industry. A popular commercial tool is Vivado HLS from Xilinx [5] [11]. This tool accepts C, System C and C++ code as inputs. The tool also requires constraints and test benches as inputs, so as to properly evaluate the implementation. Additionally, the developer can define a series of directives to control the hardware implementation, such as whether to implement loop unrolling. Vivado is based on LLVM. The inputs are compiled and then synthesized to an RTL implementation in VHDL. This RTL description is then implemented in Xilinx hardware. Vivado offers an expansive IDE to develop the hardware. This IDE allows developers to evaluate and improve the implementation on multiple levels.

Another commercial tool is MaxCompiler by Maxeler Technologies [5] [22]. This tool allows developers to build data-flow engines (DFEs) and integrate them in a CPU program. The tool

divides the application in three basic components. One are the kernels, which describe the data flow and are implemented in hardware. Another are the managers, which manage the connection between kernels and the CPU. And the rest are CPU programs to read and write from the kernels. The kernels are written using a Java based language called MaxJ. This language allows the description of data flows through objects called cores. There are multiple kinds of cores, such as input/output cores, computational cores, which implement arithmetical and logical operations, multiplexer cores for selecting many different values, etc. A dataflow is described as combination of cores that operate on an input stream and generate an output stream. This description can then be easily represented as a DFG and then compiled by the MaxCompiler to be implemented in hardware. This tool implements the DFG in a highly pipelined manner so as to achieve a high throughput. The pipelined data flow implementation the MaxCompiler tool is efficient for algorithms that allow for a fast streaming of inputs into the kernel. The tool also permits the developer to guide the hardware implementation by defining aspects like the amount of parallel DFEs.

2.3 Overview of HLS tools

Table 2.1 presents a succinct overview on some relevant aspects of the presented HLS tools. It highlights the academic or commercial nature of the tools, the input languages and the options the tools provide to users to optimize the implementation. Table 2.2 continues the overview. It presents the specific FPGA hardware each tool targets, what kind of output it generates, their compilation frameworks, and some benchmarks used in the evaluation. Some of this information is unavailable for certain tools.

| Tool or Framework | Type | Input Language | Optimizations |
|----------------------------|------------|---------------------|---------------------------------------|
| LegUP [12] | Academic | C, System C, OpenCL | Different Types of Compilation passes |
| Vivado HLS [5] [11] | Commercial | C, System C, C++ | Directives |
| Max Compiler [5] [22] | Commercial | MaxJ | Directives |
| ASC [18] | Academic | ASC | - |
| LALP [17] | Academic | LALP | - |
| Cameron Project [15] | Academic | SA-C | - |
| <i>bambu</i> [15] | - | C | Configuration File |
| Xronos for Vivado-HLS [21] | Academic | RCV-CAL | - |

Table 2.1: Overview of some of the most relevant HLS tools

| Tool or Framework | Target Devices | Output | Compiler Framework | Benchmarks |
|-----------------------|----------------|---------------------|--------------------|--|
| LegUP | various | Verilog RTL | LLVM | set of benchmarks |
| Vivado HLS | Xilinx | VHDL | LLVM based | - |
| Max Compiler | various | max file | Max Compiler | - |
| ASC | Xilinx | hardware modules | GNU compiler | Wavelet Compression, Kasumi Encryption |
| LALP | various | VHDL | LALP framework | signal processing, encoding/decoding |
| Cameron Project | various | VHDL & C (for host) | SA-C compiler | Prewitt algorithm |
| <i>bambu</i> | various | VHDL | GCC compiler | - |
| Xronos for Vivado-HLS | various | VHDL | Orcc | - |

Table 2.2: Overview of some of the most relevant HLS tools

2.4 Code restructuring

Software code is written to be implemented in a CPU, which executes most operations sequentially. As explained before, much of the acceleration from FPGAs is due to their ability to perform operations in parallel [5] [8]. However, many software programs are not written in a manner that exposes parallelism in the algorithm. This lack of clear parallelism may lead to many inefficient implementations in FPGAs. Code restructuring is necessary to generate optimized C which exposes the parallelism better than the original code.

A typical transformation is loop unrolling. In this optimization, iterations of a loop are explicitly and separately written leading to a larger loop with fewer iterations. If the iterations of the original loop are independent, then this transformation allows them to be executed in parallel in an FPGA. This is, however, a resource intensive optimization and works better for small independent loops. For larger loops it might be preferable to unroll a specific amount of iterations instead of all of them. Loop unrolling also allows the index of the loop to be described as a constant instead of a variable, meaning that the FPGA can optimize operations that use the index, like implementing a doubling as a simple logical left shift.

Another optimization is loop pipelining. In this case, the loop is restructured so that it can be easily pipelined. A way to achieve this is by loading values for the next iteration while simultaneously calculating the result of the current iteration, so that in the next clock cycle the calculations of the next output can be immediately started. This transformation allows the FPGA implementations to achieve high throughput by pipelining large loops.

However, not everything can be expressed directly in the code. Reference [10] describes how

C still lacks some requirements for hardware descriptions. There exist C-like languages such as System-C that heavily modify C to be more acceptable for hardware development. However, some tools like Vivado HLS are built specifically for typical software C. They allow the user to impose timing and resource constraints that are essential for hardware developments and they also deal with memory usage. Crucially, most tools allow users to choose different kinds of optimizations through directives. These optimizations also have drawbacks. For example, loop unrolling utilizes a lot of resources, and this is unwanted in implementations that minimize resource usage. However, if one wants to accelerate the implementation, loop unrolling is essential to increase concurrency. Thus, allowing users to choose optimizations is important, so as to achieve a desired implementation goal. However, directives have to be generic and many optimization still requires users to restructure the code. For example, Vivado HLS has a directive to unroll the loop and even implement it as a pipeline. But there is no directive that optimizes the pipeline through approaches such as the aforementioned loop pipelining technique. This improvement has to be implemented manually in the input code. Also merely unfolding might not accelerate the implementation due to memory bottlenecks, and these have to be handled by separate directives. There are other code optimizations that cannot be simply handled by directives and requires users to manually restructure the code, which can be complicated.

The article by Stephen Cong et al. [9] explains the issues of code restructuring and presents a framework that attempts to facilitate code restructuring. The article starts with a simple code with three nested loops and modifies it step by step. They change the loop order, add pragmas to handle memory, pipeline the loop, optimize memory usage and implement a first in first out (FIFO) array. The final code is significantly different and a user not used to Vivado HLS might have difficulties writing such code. To make this restructuring they present a framework, which is Merlin Compiler. Its purpose is to receive code with a few directives and generate a restructured Open CL output for a specific FPGA. This Open CL code can then be used by a HLS tool. The flow of the tool is to first parse the input code and generate an abstract syntax tree (AST). It then analyzes the AST recording information such as loops and accesses. Afterwards, the compiler models the program into an intermediate graph representation and passes it through multiple optimization passes. After optimization, it develops an interface with the software for the resulting program and generates the OpenCL code.

For optimizing the input the compiler requires a few directives. These are pipeline, parallelize and task pragmas. The last pragma is used to specify the tasks to accelerate. The parallelize and pipeline pragma are used to specify which type of optimizations to implement. These pragmas are context sensitive. If this pipeline pragma is placed on outer loops or functions calls, the compiler implements coarse grained pipelining, meaning it analyzes the contents of the loop and its dependencies and groups them into tasks. The tool then schedules the tasks into stages and connects these to be able to implement an efficient pipeline. If the parallelize pragma is placed on outer loops the compiler will attempt to implement coarse grained parallelization, meaning it instantiates units that implements the contents of the loop in parallel. If the contents of the loop are not completely independent the compiler handles the dependencies to parallelize as much as possi-

ble. If these pragmas are placed on the innermost loops the compiler optimizes the code at the instruction level. The pipeline pragma optimizes the pipelined execution of the instruction while the parallelize pragma directs the compiler to attempt to execute the contents in parallel. These directives can be combined. The compiler analyzes the instruction and restructures the code to implement these efficiently. One of the factors it focuses on is handling the memory accesses to ensure that the data necessities do not limit the initiation interval of the pipeline or parallel units by partitioning the memory into appropriate banks in the FPGA. The compiler also implements some optimizations automatically, such as reusing data. In this case the compiler keeps data read from memory that is used later in a buffer.

Another tool with a similar approach is detailed in [23]. The tool accepts a C file and an extra file that details the compilation strategies. This file is written in an aspect oriented language called LARA. This language allows for the developer to define compilation strategies independently that can be ported to multiple different codes as well as non functional constraints that are important for hardware. The authors implement a strategy by defining three sections of an aspect. The first is the *select* section in which the authors define the segment of code that they wish to target, such as a statement, a function or a type of loop. When compiling the code the tool places all the segments that were selected for the aspect in a table. Thus, if the user selects all innermost loops, the tool builds a table with all innermost loops. Another section is the condition section, in which the developer can place conditions on the types of elements selected. So if all loops are selected, this section can limit the selection to loops of certain sizes. The final stage is the *action* stage, in which the authors detail what the optimization the tool might do for the selected code segments that fulfill the conditions. This can be modifications to the code or the additions of pragmas. The input source code is passed through a weaver. The weaver handles the partitioning of the input for hardware/software implementation as well as the communications between these two parts. There is also an IR weaver in which optimizations are made to an IR representation of the input. After the optimizations, the resulting C code is fed into an HLS tool. The framework also allows for feedback from the synthesized implementation as well as profiling the input source code. The tool implements a design exploration tool that receives the feedback and allows for changes in the input LARA file.

The authors detail actual optimizations done to two specific case studies. One is a 3D path-planning (3DPP) which is an application that plans a path for a flying vehicle, and the other is stereo navigation that uses two images to inform a vehicle about its navigation. For 3DPP profiling indicates that 90% of the execution time is dedicated to a function called *gridit* which has 3 levels of nested loops that update a 3d matrix. One of the first optimizations is done to the loop that calls the *gridit* function. This loop calls *gridit* 5 times and has 12 iterations. This loop can be moved inside *gridit*. So, instead of having 60 calls of *gridit* the implementation has only 5, and the function has an added outer loop with 12 iterations. The algorithm still does the same number of total iterations but the invocations of *gridit* are far fewer which vastly improves SW/HW systems, because there is a lot of overhead associated with communicating with a function in hardware. Afterwards, they unroll the innermost loop and replicate a matrix to have multiple concurrent load/stores. They also

coalesce the middle loops since no action is done between them so coalescing increases efficiency of the execution. The authors also use on-chip memories to reduce the number of data transfers as well as reuse data to minimize accesses. So these are multiple different types of optimizations that are more intuitive to the hardware specialist. A lot of these optimizations such as the loop unrolling are implemented through pragmas. In this case the target HLS tool is Catapult-C, therefore the authors use its directives.

As for the second case study the heaviest execution time is dedicated to convolutions implemented through three functions with multiple single precision floating points and accumulations. To improve the execution the authors implement a strategy that executes the convolution operations in parallel. Floating point operations are harder to implement in HW than integer-based operations, so the authors attempt to implement any floating point as an integer if possible. Inputs that are constants are instead implemented in local arrays in HW to minimize data transfer.

When comparing these optimized version with the software implementations the authors obtained high speed gains. In the case of 3DPP, the hardware cores implement their tasks with an overall 12.15x speedup compared to the software version, and the overall application has a 6.8x execution speed gain. As for the second case study the speedup of the overall execution is 2.12x and the convolution is 3.90x.

Another in depth case study that demonstrates the impact of code restructuring is discussed in [24]. The authors present a HW/SW FPGA implementation for an application that detects arrhythmia through digital signal processing of electrocardiogram signals using Vivado HLS. The application consists of an initial stage that detects the heartbeats and a second stage that makes a diagnostic based on the heartbeats. The diagnostic part is based on a support vector machine (SVM)[25] and it is the stage that takes most execution time. Therefore, it was the stage considered for hardware acceleration. The SVM classifier takes as an input a test vector and classifies it into one of two categories. The algorithm calculates the quadratic Euclidean distance of every input vector to a set of support vectors, each vector belonging to one of the two categories. Then every distance is weighted by a variable gamma and the exponential of the negative result is calculated. The algorithm then multiplies this with a coefficient distinct for every input vector and finally subtracts by a bias value. This calculation is done for every input vector and the results are accumulated. The algorithm then classifies the inputs based on the sign of the accumulation.

SVM is a generic algorithm and the user can define the gamma and b parameters, the support vectors and coefficients to tailor the algorithm to the desired application. The algorithm in C code consist of a nested loop that goes through every support vector and accumulates the distances. The distances are calculated in the inner loop. The first optimization is partitioning the calculations of the distances of the vectors since these are independent of each other. This independence means that those tasks can run concurrently. To do so the authors write a version of the initial SVM that uses smaller inputs and then partition the inputs so that the tasks can be called concurrently. The authors called the smaller version multiple times and utilize the HLS dataflow directive to direct Vivado HLS to run all of the tasks concurrently. The authors compare partitioning the memories manually to using directives and conclude that manually partitioned arrays are preferable. After-

wards, the authors optimize the actual SVM algorithm. They first manually unfold 6 iterations of the inner loop. This inner loop contains a chain of accumulations of all the quadratic distances. This chain lowers the ILP because it forces the additions to be scheduled sequentially. Additionally, to pipeline the inner loop this chain will increase the initiation interval of the pipeline since the implementation needs the result of the previous chain to start the next. But this is unnecessary due to the associative property of additions allowing us to add items in a different order. So the authors implement the accumulations chain as a tree to increase the ILP. After this change the authors no longer restructure that code and instead only apply directives. The applied directives are the pipeline directive, to increase the throughput of the loops, the unroll directive to increase the ILP and array reshape and array partition to increase the memory throughput.

As there are many possible combinations of these directives, the authors isolate each and execute an exhaustive search to find the best optimizations. Pipelining and unrolling the inner loop produces improvements on average. Pipelining the outer loop improves the latency, but forces a very large usage of resources because the directives forces the inner loop to be fully unrolled. The memory directives lead to large increase in block RAM usage and have a lower impact on the usage of other resources and the latency. The authors outline afterwards a guideline to setting directives. They advise that the partition factor or reshape factor if used alone must be equal to the unroll factor of the inner loop. If used together the product of their factors must equal that of the unroll factor. Using these guidelines they vastly lower the search space for the directives. Utilizing the best combination of these, the authors achieve a performance in hardware that is 78.9 times better than the original. They also compare their implementation with full SW implementations and conclude that they obtain a 10 times increase in performance compared to a dual-core 64 bit ARM CPU.

There have been other attempts to detail the impact of correct code optimization. In [26] the authors use LegUp to study the impact of compiler optimizations on performance of HW implementations. LegUp, as explained before, implements compilation passes to optimize the code for HLS. The authors analyzed the impact of individual passes and combinations of these. They also offer a method that generates custom recipes of passes and show how these custom recipes perform better than the standard LegUp passes. The paper also analyzes what kind and in which situations certain code optimizations have a significant impact on performance. It determines that the most useful passes are those that increase ILP, remove operations or allow further optimizations. This article reiterates the necessity of restructuring the code, and how that in combination with an HLS tool directives can achieve better results.

Efficient off-chip memory usage is an important issue in the use of hardware accelerators. Many implementations do not take into account some issues with memory such as cache misses. Therefore, such problems can lead to many wasted clock cycles as the execution stalls to obtain the correct memory value. Some approaches to improve this problem have been presented. In [27] the authors present an approach that transforms CDFGs before inputting them to an HLS tool. The idea of the approach is to generate a dataflow engine from the original CDFG. The original CDFG is analyzed and partitioned into multiple stages of a pipeline. A new pipeline stage is created when

there is a memory access or the latency of a node is large. This partitioning allows for latency due to memory issues to be hidden, since cache misses happen while other stages are being executed. Thus, instead of the execution being stalled, the other stages proceed independently. As long as the value is obtained before another stage requires it, the execution is not stalled. This approach also allows for the different stages to be optimized independently.

2.5 Graph restructuring

Dataflow graphs are highly used in HLS tools since they are suitable representations for hardware models [8]. A dataflow graph is a directed graph that models the way data changes when flowing from the input to the outputs. The nodes of the graph typically represent basic operations, although for higher levels of abstraction they can also represent a more complex function. The edges connecting the nodes model the way data flows through the system. This representation is appropriate since the nodes can be directly associated to hardware units in an FPGA. They are also capable of explicitly representing parallelism. If sections of the data flow are not dependent they can then run concurrently. A well optimized graph can greatly improve speed and resource usage. Due to their utility in hardware development there are several studies of using DFGs for FPGAs and what kind of graph leads to a better hardware implementation.

N.Voss et al detail in [28] many important optimizations for DFGs. Their goal is to present a method of automatically generating an optimized graph focusing on datapath merging, which is relevant for our work. The authors present a tool that implements the optimizations automatically. It uses the Max-Compiler which was previously mentioned. Max-Compiler represent every kernel in the algorithm as a DFG. The authors' approach combines every graph in order to optimize the entire dataflow at once. Then they eliminate unnecessary code. They proceed with eliminating unnecessary operation through constant folding. Associative operations are also optimized. If two similar ones are connected they can be combined into a single node minimizing resource usage. If the dataflow has a sequential chain of associative operations the chain can be compacted into a balanced tree which can result in large improvements due to the increased parallelism and faster output generation. To further optimize the DFG the tool eliminates redundant operations. If the DFG executes two equal operations with the same inputs then the results are going to be same, therefore duplicated nodes are removed. Thanks to the previous associative node merging the tool can recognize more hidden redundant operations. The calculations $a + c + b + e$ and $e + c + a + b$ are redundant but without node merging this factor is more difficult to discover.

The tool attempts to minimize divisions which are very costly in hardware by taking advantage of operations that have common divisors. If that happens, the numerators are first multiplied with each other and the result is divided by the common divisor. This improves the implementation, because it uses more multiplications than divisions. Similarly, another optimization is to substitute divisions by constants with multiplications with the inverse value of the constant.

Another important element is merging identical sequences of nodes. The authors detail the merging conditions, which are that the nodes need to be of the same operations and handle the

same type of input data. The authors also need to be careful merging nodes to avoid a situation in which a node needs to calculate two results in the same cycle which might not be possible. The tool distinguishes inputs of merged nodes using multiplexers. Therefore, it is unnecessary to optimize operations, such as *and* or *or* since they use less resources in hardware than multiplexers. The decision about which nodes to merge is via a heuristic. A merger adds new dependencies to the graph and also changes it creating potential new merges. The heuristic is greedy and will attempt to first merge nodes in order of expensiveness. The tool starts with most expensive one. The tool determines a list of candidates based on the shared inputs and proximity. Multiple mergers can lead to a large number of large multiplexers, so the tool minimizes the number of multiplexers and their size. These optimizations can greatly improve the resource usage of the graph and the authors present two examples in which they achieve 2x and 4x less area.

Another very important aspect is matching sequences of nodes, thereby identifying large sequences which can be manipulated in various ways to accelerate execution of the program. The authors of the article [29] demonstrate a design flow to partition a DFG into multiple subgraphs by taking advantage of isomorphism in the graph. They attempt to identify large identical clusters in an algorithm, and afterwards accelerate their execution in hardware. The matching algorithm they propose is divided into four parts. In the first stage they level the graph and assign a weight to every node based on a heuristic that accounts for the level the node type and the nodes it is connected to. The weight algorithm assigns the same weight to similar nodes. Leveling is a process by which each node is identified by its depth in the graph. Next, the algorithm identifies all possible subgraphs. The algorithm does an exhaustive search of the graph, level by level. For every node of a level the algorithm creates a subgraph. The algorithm builds up the subgraph by adding adjacent nodes to the first node added in the subgraph. It continues building the subgraph by checking the nodes connected to every new node added to the subgraph. A node is added if its level is neither lower than that of the original or nor higher than the current highest level of the subgraph plus 1. If there are no more nodes that meet the criteria, the algorithm adds the subgraph to a list of subgraphs. After applying this to every node of a level, the algorithm advances to the next level. This process ends once the algorithm reaches the final node of the last level. In the next stage all subgraphs are given a weight and compared with each other. Subgraphs of similar weight are put on a list of isomorphic graphs, and the performance of the subgraphs is estimated. The algorithm is applied to benchmarks from digital signal processing since they have many isomorphic sequences.

The approach of this dissertation also handles matching in the graph to be able to compact the extremely large graphs that represent execution traces. Although the purpose is different compared to the one explained, the authors demonstrate a manner to match patterns in a DFG, and the impact of this merging on the type of algorithms that are also targeted by our work.

2.6 Summary

This chapter presents the related work in the fields that are essential for this dissertation. The first section discusses various HLS tools to understand the state of the art of the available tool. There are many different ones with many distinct approaches. An overview of the state of the art of HLS tools indicates that a lot of new approaches are being tested, and there are already implementations with positive results. This chapter also presents literature related to restructuring C code for HLS tools. Some specific examples of code restructure and the obtained result are detailed. Additionally, the chapter presents some tools that attempt to implement such restructuring automatically. The final sections discuss work related to handling and optimizing DFGs specifically for FPGA implementations.

The survey [5] at the beginning of this chapter shows how active the field of HLS tools is. Although, in this chapter we highlighted some that more connected to our approach, the survey shows a multitude of current tools with various approaches, showing there is space for innovation in the field of HLS tools. Among many tools the C programming language is one of the possible input languages. However despite the popularity of C it is not a language that perfectly fits descriptions for HLS, as it is meant for sequential software implementations. Therefore, many tools have approaches to identify characteristics in the code that are essential for good hardware implementations. The most important is concurrency. By implementing multiple operations simultaneously the hardware implementation can outperform software implementations. Concurrency can be achieved by performing multiple different operation in parallel or identifying segments in the algorithm, such as loops that can be implemented as pipelines. The studied works show that to expose such properties in the C code it is necessary to restructure it, as well as use some specific directives or commands from HLS tool. In the case of Vivado HLS the latter are directives injected in the code. DFG are a good representation of hardware models and by optimizing them we can ensure an improved hardware performance. Thus, there are ways to study DFGs of algorithms to generate C code that has the aforementioned properties more exposed.

Chapter 3

Description of the Approach

This chapter gives an overview of the developed framework to generate optimized C code with directives targeting Vivado HLS.

3.1 Overview of the frontend

The purpose of the frontend of our framework is to build a DFG representing the execution trace of an algorithm. The DFG includes every operation from the original execution and data dependencies are obligatorily maintained. This last aspect is important as the graph only records the dependencies of the algorithm and not the actual execution order, therefore operations that can execute in parallel appear without interdependencies in the DFG. The frontend is intended to be as simple and generic as possible to in order fit many different inputs. The initial frontend was implemented for C code input, but it can be easily be ported to other programming languages. The DFG is generated by inserting instrumentation code in the original software code of the input. Currently, the instrumentation code is inserted manually. Figure 3.1 shows the structure of the frontend of the framework.

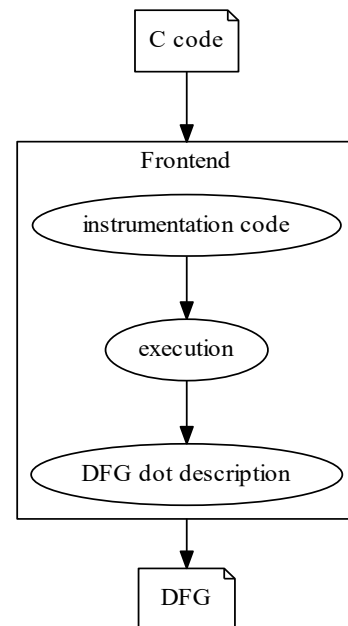


Figure 3.1: Architecture of the frontend

3.2 DFG at the frontend

The DFG generated at our frontend is a directed acyclic graph that is structured to represent an ex-

ecution trace of the algorithm. As the name indicates, dataflow graphs describe how the input data changes throughout the execution resulting in the output values. Currently, the DFG uses three types of nodes to describe the flow of data of the targeted C code. These are the constant nodes ("const") that represent constants used in the code, the variable nodes ("var") that represent the construct of variables in the C language. The frontend DFG can represent scalar variables and arrays with these nodes. "var" nodes represent the value in a variable at a specific point of the execution. If the value in a variable is changed, a new node to represent the variable with that value is added to the DFG. Thus, the DFG can contain many nodes representing the same variable, as they are all associated to points during the execution where that variable was written, thereby representing how data changes throughout the execution of the code. In addition there are the operation nodes ("op") that represent the operations in the C code. Currently, the frontend DFG handles arithmetic and logical operations such as $+$, $-$ or \ll . Edges represent the data transfer between nodes. The entering edge of a "var" node represents the storing of data into the variable, and the outgoing edge represents the transfer of the data in the variable. The entering edges of the "op" node connect it to the data that is used in the operation, and the outgoing edge represents the transferring of the resulting value. By connecting these nodes, the resulting graph shows the way data is changed throughout a kernel's execution. Figure 3.2 shows the DFG resulting of the statements $b = a$ and $d = c + b$. The input data starts in a and c . The data in a is stored in variable b . The data from c and b is inputted in to the "op" node $+$ that represents the sum in the code. Finally that resulting data is stored in d that is the output of the code.

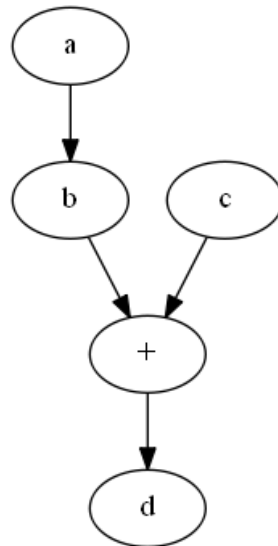


Figure 3.2: Graph corresponding to statements $a = b$ and $d = b + c$

3.3 Operation descriptions

The output DFG is described in the DOT language [30]. In this language nodes are described by an identifier and a sequence of attributes (see Listing 3.1). Every node has at least two attributes.

| Node type | Attributes |
|--------------|--|
| "op" Node | label: stores the symbol of the operation att1: op |
| "var" Node | label: store name of variable att1: var att2: locality of the variable att3: type of variable |
| "const" Node | label: value of constant att1: const |

Table 3.1: Overview of frontend node information

One is the label and the other the type. A node can have three types, which are constant ("const"), variable ("var") and operation ("op"). As for the label attribute, it stores the name of the variable, the value of a constant or the symbol of an operation depending on the type of node. If a variable is an array the node also includes the index of the access. Additionally, variable nodes also hold as attributes the type of variable and whether it is a local variable or an input parameter of a function.

Currently, the framework can only deal with kernels consisting of basic operations. Assignments are represented by connections between nodes (e.g., from a constant type node to a variable type node). Operations are represented by "op" nodes connected to "var" and "const" nodes. The nodes connected to the entering edges of the "op" node are the operands and the node connected to the outgoing edge is the result of the operation. The result node can only be a "var" node as constants and other operations cannot be a result of an operation. Additionally, the front end also supports function calls that modify a value of a variable. These modifications are added as an attribute to an edge. So for the case of $b = \text{abs}(a)$ the edge that goes from a to b has as an added attribute called *mod* containing the value *abs()*. The approach does not have an equivalent operation node for a return statement. Instead, a node of the same type is instantiated and defined as an interface node. Thus, the equivalent operation for the return statements is assigning the returned variable to the added interface node.

As a simple example, Listing 3.1 shows how to describe in DOT language the DFG in Figure 3.2. The description starts of specifying that the graph is directed with the statement "Digraph G". Afterwards, all the nodes are described. In this case there are 4 "var" nodes that are all of type *int* and they are all local variables. These nodes represent the variables a , b , c and d . We have the edge connecting a to $\text{textit{b}}$ to represent the transferring of data from the $b = a$ statement. Afterwards, the three edges used to connect the nodes for the statement $d = b + c$ are declared. The edges connecting the operand b and c have as attributes the side of the operand in the original code.

```
Digraph G{
a [ label = a , att1=var , att2=loc , att3= int ]    \\ nodes
b [ label = b , att1=var , att2=loc , att3= int ]
c [ label = c , att1=var , att2=loc , att3= int ]
d [ label = d , att1=var , att2=loc , att3= int ]
```

```

+ [label = +, att1=op]
a->b                                \\ edges
c->+ [pos=r]
b->+ [pos=l]
+>d
}

```

Listing 3.1: Basic graph described in the DOT language

3.3.1 Graph generation

The current approach to the generation of the DFG, representing the execution of a kernel, is to write out the DOT description to a file during execution. The DFG is generated by injecting instrumentation code into the original C version, compile and executing it. The basic instrumentation rule is to append the code before statements in the original C code. The added code describes the operations based on the approach described earlier. For an assignment statement the input "var" or "const" node and the output "var" node are instantiated. To complete the description of the assignment, the instrumentation code to describe an edge from the input node to the output node is added. For other statements the approach prints out four nodes. The values of the nodes describe the current operation, so for a statement such as $a = b + c$ the instrumentation code generates a description in the *DOT* language of a node for the variables c , b and a , and includes as attributes the variable names, type and locality. If a node is an array access, the added instrumentation code describes it as a variable and the label is the access with the explicit array index. The operation node in the statement $a = b + c$ has the label $+$. The description of the statement is completed by adding edges from the operand nodes to the "op" node, and an edge from "op" to the result node. The edges that connect the inputs to the operation node also record the position in the statement, in this case right for b and left for c . An "op" node can only have two input nodes. If the code has a single statement with multiple operations, such as $a = b + c + d$, it is decomposed into $temp = c + d$ and $a = temp + b$. Thus, the approach defines an extra variable node to hold the value of the first calculation, and this is used in the next operation. It is important to ensure that the name of the variable is unique, so that the resulting code is correct since these are all new variables. To ensure the uniqueness, the labels for these extra variables are the word "temp" appended with the number of the line of original statement in the code. In case the variable is added in a loop, the iteration number is also appended to further differentiate the variable. So if we had in a loop the statement $a = b + c + d$ at line 5 in the code, the names of the added variables would be temp_15_i1, temp_15_i2, temp_15_i3 and so forth.

Another very important aspect is to ensure the correct representation of dependencies. A variable can have multiple values along the execution. For every new value a variable takes, a new node has to be created in the graph. The tool must also be able to know exactly the identifier of the node that corresponds to the most recent value of a variable instead of any outdated value. The instrumented code uses as identifiers a modified version of a variable's name to reflect the most

recent assignment. The instrumentation code adds to the variable name a number representing that assignment. So, if the code starts with the sequence $a = b$; $d = a + c$; $a = d + a$; the identifiers a_1 b_0 for the first operation d_1 a_1 c_0 for the second and a_2 d_1 a_1 for the last are used. The approach also uses a counter for the number of "op", extra variables and "const" nodes created. When a new operating node is created the counter is incremented. The identifier of the "op" node is the word "op" append with the counter number, to ensure that all "op" nodes are unique. The same process is applied for the "const" and the added variables.

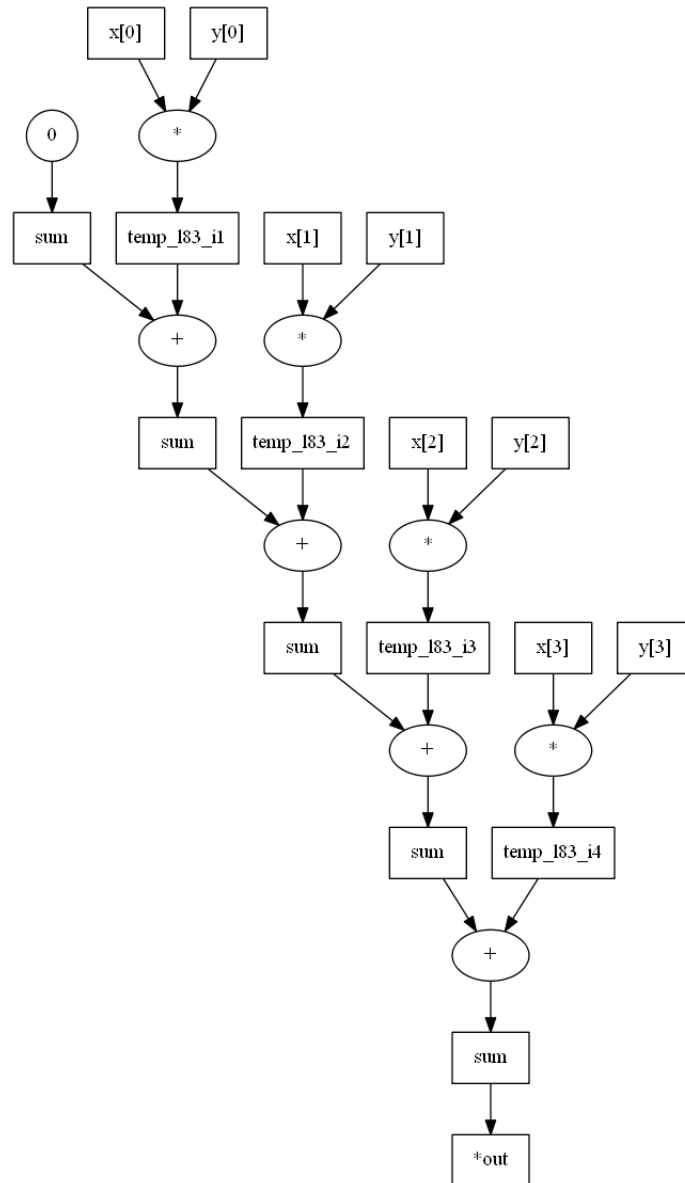


Figure 3.3: DFG of the dot product with vectors of size 4

Listing 3.3 shows the *dotproduct* benchmark (see original in Listing 3.2) code with the instrumentation code. Listing 3.4 shows the DFG resulting from the execution of the code in Listing 3.3 with a vector of size 4. Since this frontend was applied to C code the text was written to a dot

file using the *fprintf* function. In this case the output file is "dotprod_graph.dot". The first string written ("Digraph G") specifies the name of the graph and that it is directed. At the beginning, the instrumentation code to initialize all the counters is added. Then, the instrumentation code for each operation is added. For the first operation $sum = 0$ the two necessary nodes are created. The constant the counter is incremented in line 14 and in line 15 the string describing the "const" node is printed out to the output file. As per our approach the identifier is the word "const" appended with the counter and the label is the value of the a constant and the type of node is "const". Line 16 prints out the description for the variable *sum*. As the *sum* variable is the result, the identifier of the node is the name of the variable appended with the counter incremented by one. The attribute label is "sum", as that is the name of the variable. The rest of the attributes specify that it is a "var" node and that the variable is of type *int* and local. Then, in line 19 the edge connecting the two nodes is printed and afterwards the counter for the *sum* variable is incremented. These operations generate in the dot file the lines 4, 5 and 6.

Through lines 28 to 67 the instrumentation code of the statement $sum += x[i] * y[i]$ is shown. Lines 28 to 31 generate the nodes describing the variables $x[i]$ and $y[i]$. We can see in those lines that the label of the nodes use the exact index access. The instrumentation code in the next 5 lines describe the "op", which in this case is a multiplication, and the result node. The node that receives the result of the operation is in this case an extra variable to break up the statement, according to the approach we described earlier. Then, the code that instantiates the respective edges is added. Lines 10 to 17 of Listing 3.4 show the description of nodes and edges that represent the multiplication for the first iteration. Lines 50 to 67 of Listing 3.3 show the instrumentation code for the addition of the statement. In this case we have two nodes that represent the same variable but with two different values. The statement the instrumentation code represents is $sum = sum + temp$. Thus, one of the input nodes of the operation should link to the last value of the variable *sum* and a new node should represent the new value of the variable *sum*. So, the instrumentation code in Line 52 generates a node with the identifier *sum* appended with the current value of the counter for the variable *sum*, while the instrumentation code in Line 56 generates a node with the identifier *sum* appended with the incremented value of the counter for the variable *sum*. Lines 18 and 20 in Listing 3.4 show the description of these two nodes for the first iteration. The first node has the identifier *sum_1* and the other *sum_2*. Line 5 shows that the previous node for the variable *sum* has the identifier *sum_1*, so the node description coincides as required. Figure 3.3 shows the graph described by the dot in Listing 3.4. The DFG clearly shows the dataflow of the dot product of two vectors. Figure 3.3 presents multiple nodes with the label *sum*. All these different nodes represent the values that the variable *sum* takes along the execution of the program.

```
int dotprod(const short *x, const short *y) {
    int sum=0;
    for (i = 0; i < NX; i++) {
        sum += x[i] * y[i]; }
    return sum; }
```

Listing 3.2: Code of the *dotproduct* benchmark

```

1  int dotprod_graph(const short *x, const short *y)
2  {
3      short n_x[NX]={0};
4      short n_y[NX]={0};
5      short n_sum=0;
6      int n_const=0;
7      int n_temp=0;
8      int n_op=0;
9      int n_out=0;
10     int ne=0;
11     FILE *f=fopen("dotprod_graph.dot","w");
12     fprintf(f,"Digraph G{\n");
13
14     // instrumenation for assignement
15     n_const++;
16     fprintf(f,"const%d [label=\"0\", att1=const];\n",n_const);
17     fprintf(f,"\"sum_%d\" [label=\"sum\", att1=var, att2=loc,
18         att3=int ];\n",n_sum+1);
19     ne++;
20     fprintf(f,"const%d->\"sum_%d\" [ ord=\"%d\"]; \n",n_const,n_sum+1,ne,ne);
21     n_sum++;
22     int sum=0;
23
24
25     for (i = 0; i < NX; i++){
26
27         // instrumentation code for the multiplication
28
29         fprintf(f,"\"x[%d]_d_l\" [label=\"x[%d]\", att1=var, att2=inte,
30             att3=short ];\n",i, n_x[i] ,i);
31         fprintf(f,"\"y[%d]_d_r\" [label=\"y[%d]\", att1=var, att2=inte,
32             att3=short ];\n",i, n_y[i] ,i);
33         n_op++;
34         fprintf(f,"op%d [label=\"*\", att1=op];\n",n_op);
35         n_temp++;
36         fprintf(f,"temp%d [label=\"temp_l83_i%d\", att1=var, att2=loc,
37             att3=int ];\n",n_temp,n_temp);
38
39         ne++;
40         fprintf(f,"\"x[%d]_d_l\"->op%d [ord=\"%d\", pos=\"l\"]; \n",
41             i, n_x[i],n_op,ne,ne);
42         ne++;
43         fprintf(f,"\"y[%d]_d_r\"->op%d [ ord=\"%d\", pos=\"r\"]; \n",
44             i,n_y[i],n_op,ne,ne);

```

```

45     ne++;
46     fprintf(f,"op%d->temp%d [ ord=\"%d\"];\n"
47         ,n_op,n_temp,ne,ne);
48
49     // instrumentation code for the sum
50
51     fprintf(f,"temp%d [label=\"temp_l83_i%d\", att1=var, att2=loc,
52         att3=int];\n",n_temp,n_temp); //nc-2
53     fprintf(f,"\"sum_%d\" [label=sum, att1=var, att2=loc,
54         att3=int ];\n",n_sum);
55     n_op++;
56     fprintf(f,"op%d [label=\"+\", att1=op ];\n",n_op); //nc-1
57     fprintf(f,"\"sum_%d\" [label=sum, att1=var, att2=loc,
58         att3=int ];\n",n_sum+1);
59
60     ne++;
61     fprintf(f,"temp%d->op%d [ ord=\"%d\", pos=\"r\"];\n",
62         n_temp,n_op,ne,ne);
63     ne++;
64     fprintf(f,"\"sum_%d\"->op%d [ ord=\"%d\", pos=\"l\"];\n",
65         n_sum,n_op,ne,ne);
66     ne++;
67     fprintf(f,"op%d->\"sum_%d\" [ ord=\"%d\"];\n",n_op,n_sum+1,ne,ne);
68     n_sum++;
69
70     sum += x[i] * y[i];
71 }
72
73 // instrumentation code for the output
74 fprintf(f,"\"sum_%d\" [label=sum, att1=var, att2=loc,
75     att3=int];\n",n_sum);
76 fprintf(f,"\"out_%d\" [label=\"*out\",att1=var, att2=inte,
77     att3=int];\n",n_out);
78 fprintf(f,"\"sum_%d\"->out_%d [ ord=\"%d\",pos=equal];\n",
79     n_sum,n_out,ne,ne);
80 return sum
81 }

```

Listing 3.3: Code of dot product with instrumentation code

```

1 Digraph G{
2     \\ sum=0
3
4     const1 [label="0", att1=const];
5     "sum_1" [label="sum", att1=var, att2=loc, att3=int ];
6     const1->"sum_1" [ ord="1"];

```

```

7
8      \sum+=x[0]*y[0]
9
10     "x[0]_0_l" [label="x[0]", att1=var, att2=inte, att3=short ];
11     "y[0]_0_r" [label="y[0]", att1=var, att2=inte, att3=short ];
12     op1 [label="*", att1=op];
13     temp1 [label="temp_l83_i1", att1=var, att2=loc, att3=int ];
14     "x[0]_0_l"->op1 [ ord="2", pos="l"];
15     "y[0]_0_r"->op1 [ ord="3", pos="r"];
16     op1->temp1 [ ord="4"];
17     temp1 [label="temp_l83_i1", att1=var, att2=loc, att3=int ];
18     "sum_1" [label=sum, att1=var, att2=loc, att3=int ];
19     op2 [label="+", att1=op ];
20     "sum_2" [label=sum, att1=var, att2=loc, att3=int ];
21     temp1->op2 [ ord="5", pos="r"];
22     "sum_1"->op2 [ ord="6", pos="l"];
23     op2->"sum_2" [ ord="7"];
24     ...
25     ...
26     \sum+=x[4]*y[4]
27
28     "x[4]_0_l" [label="x[4]", att1=var, att2=inte, att3=short ];
29     "y[4]_0_r" [label="y[4]", att1=var, att2=inte, att3=short ];
30     op9 [label="*", att1=op];
31     temp5 [label="temp_l83_i5", att1=var, att2=loc, att3=int ];
32     "x[4]_0_l"->op9 [ ord="26", pos="l"];
33     "y[4]_0_r"->op9 [ ord="27", pos="r"];
34     op9->temp5 [ ord="28"];
35     temp5 [label="temp_l83_i5", att1=var, att2=loc, att3=int ];
36     "sum_5" [label=sum, att1=var, att2=loc, att3=int ];
37     op10 [label="+", att1=op ];
38     "sum_6" [label=sum, att1=var, att2=loc, att3=int ];
39     temp5->op10 [ ord="29", pos="r"];
40     "sum_5"->op10 [ ord="30", pos="l"];
41     op10->"sum_6" [ ord="31"];
42
43     \return sum
44
45     "sum_6" [label=sum, att1=var, att2=loc, att3=int ];
46     "out_0" [label="*out", att1=var, att2=inte, att3=int ];
47     "sum_6"->out_0 [ ord="32", pos=equal];
48 }

```

Listing 3.4: Dot description of the *dotproduct* DFG with vectors of size 4

3.3.2 Information lost through tracing

This approach for generating DFGs has an important issue concerning information compared to the original C code. As described, our approach represents simply the flow of data during the execution. Thus, the approach does not have a method of representing constructs such as *for* or *while* statements at the frontend. The goal of the frontend is to generate the resulting unfolded DFG and input into the backend, so that it can restructure this basic representation of the original kernel. This leads to a problem if kernel depends on certain inputs and calculations made inside a loop. For example, Listing 3.5 shows three different kernels. The execution trace of the first kernel would be composed of 3 increments to the input variable *a*. Thus, the trace would be valid for any input of the kernel. However, for the second kernel the *for* loop depends on the input variable *b*. If the execution was traced for *a b* equaling 5 the resulting trace would have 5 increments of *a*. So, the backend would generate a C code that always increments *a* five times regardless of the value of *b*. Thus, the resulting C code of the kernel is only valid for implementations of *b* equaling 5. In the third kernel the loop depends on both of the inputs so the resulting trace would only be valid for certain combination of inputs. Therefore, when implementing this approach at the frontend, it is essential to understand which information is going to be lost in the creation of the DFG, to be sure that the resulting C code will be usable.

```
void main() {
    int a,a1,a2,a3,b;
    a1=foo_kernel_1(a);
    a2=foo_kernel_2(a,b);
    a3=foo_kernel_3(a,b);
}

int foo_kernel_1(int a) {
    for(int i=0; i < 3, i++)
        a+=1;
    return a;
}

int foo_kernel_1(int a,int b) {
    for(int i=0; i < b, i++)
        a+=1;
    return a;
}

int foo_kernel_1(int a,int b, int N) {
    for(int i=0; i < b-a, i++)
        a+=1;
    return a;
}
```

Listing 3.5: Examples of kernels

Possible uses of hardware accelerators generated for the two latter examples would have to evaluate the validity of them according to the input values and to decide at runtime about their use or not.

3.3.3 Fronted limitations

The approach has currently some limitations and it was out of scope of this dissertation to address those limitations. For example, conditional statements such as *if* or *switch* statements are a type of construct our framework does not currently handle. Since conditional statements change the flow of the execution it is not trivial to include them in the DFG. Potentially the approach could ignore the conditional statement, but the DFG would not correctly represent the code, instead the DFG would only represent the execution for the given inputs. If the conditional statements branch into different dataflow paths, then with this approach only one of the paths would be represented in the resulting DFG. Meaning that the backend could optimize the DFG, but the resulting C code would only be correct for specific input values. Another option would be describing all the potential dataflows, but then that would go against the original idea of representing the execution trace, since the resulting dataflow would include operations that did not in fact occur in the original sequence. Therefore, to represent such conditional statements there would need to be a decision regarding the nature of the input DFG in this approach. Currently, the frontend simply does not handle conditional statements.

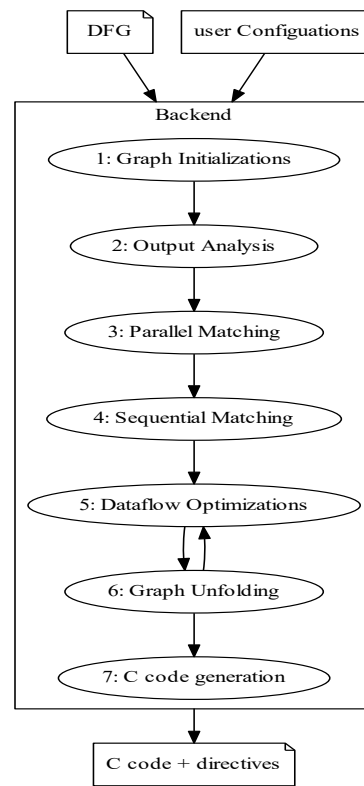
Another issue is handling accesses that depend on inputs. As stated beforehand as the execution is traced the accesses are recorded with their index. Meaning the DFG does not store the information on the calculation that generates this input. Therefore if the access depends on information that cannot be determined solely from the graph, then optimizations in the backend might not be applied. Therefore, a programmer that uses this approach has to be sure that vector accesses are independent of the input values, thereby making the final code non applicable for other inputs. A potential solution to this problem would be implementing the access to an array as an operation instead of a single variable node. This "op" node would symbolizes an array access, and the operands would be the array and the index. The index could be a constant or a variable. This approach would allow for a more versatile representation of array accesses compared to the current approach.

Function calls also present limitations for the approach. Currently, the approach is capable of representing functions that modify the value of a single variable, but this implementation is still very limited. Currently, the execution of the called functions is not traced and therefore cannot be optimized by the backend. Also if functions have multiple inputs they cannot be represented solely as an attribute in an edge. Additionally, there is also the issue of the inputs being changed inside of the function in ways that cannot be determined without checking the code of the function, since functions do not explicitly state which inputs are changed. For example, the function $foo(a, b, x)$ can change the value of a , and that is not clear simply by the function call. Thus, without knowing the content of foo the next statements that use a would not depend on foo and the trace would be incorrect. A solution could be the representation of the function as a node. This would allow

for the representation of functions with multiple inputs. However, the node would also need a way to store a representation of the trace of the functions content. That would require adding instrumentation code to the called function. Additionally, the node would also need to store the information about which input is changed to record the correct interdependencies in the execution trace. Thus, the implementation of function calls is not trivial with this approach.

3.4 Overview of backend

The framework consists of stages responsible for analyzing and optimizing the input DFG. The purpose of the backend is to optimize the DFG to generate optimized C code with directives that can be input to an HLS tool. The exact optimizations depend on the input DFG and the users given configurations. The user can choose the number of simultaneous load/stores, whether to optimize memory accesses and arithmetic operations. The user can also choose to avoid optimizations, although that will generate as output a code comprised of all the instructions in the algorithm fully unfolded. If the algorithm has many instructions then the output without graph folding will be unfeasible to implement in an FPGA due to resource demand. Appendix B shows an example of a possible configuration. The backend is the heaviest and most complex part of the framework as it implements all the code restructuring, optimizations and injects the directives for Vivado HLS.



3.4.1 Backend graph description

Sections 3.3 and 3.2 detailed the properties of the input DFG. During the course of the backend the input DFG is modified and updated. One of these modifications is the node types. The backend expands on the type of nodes so that it can achieve a graph with the desired properties of the approach. The first new node type is the "nop" type. This type is used to signal the nodes that are used to make connections in the graph but do not represent any variable, structure or operation like other nodes. The most important "nop" nodes are the Start and End nodes. As their name suggests, these nodes signal the start and end of a dataflow and are essential for the analysis of the graph.

Figure 3.4: Architecture of the backend

| Node type | Attributes |
|--------------|---|
| "nop" Node | label: either start or end att1: nop |
| "hyper" Node | label: hyper att1: hyper subgraph: stores the subgraph type: indicates the type of subgraph stored |
| "buff" Node | label: buff att1: buff |

Table 3.2: Overview of backend node information

It also leads to every graph having a similar structure with a single starting and ending node. The terms Start and End to refer to these nodes will be used from herein.

Another type are the "buff" nodes, which exist to establish a certain sequence order in the graph. In certain situations there are operations that have to follow a specific order in the C code but that order is not explicit in the dataflow. By connecting operations with a "buff" node correct sequencing is ensured. The nodes can be seen as a specific "nop" type but they are differentiated so that they can be handled differently.

The final new node type is the "hyper" type. As explained before, the graphs express dataflows, but some of these happen in different contexts such as loops. The transition between these different contexts is handled by a hierarchical representation of the dataflow. Traversing to lower levels of the hierarchy is signaled by the "hyper" nodes. These nodes contain as an attribute a graph that describes the new dataflow. This graph has attributes, which inform the context to interpret the graph. Lower levels can have "hyper" nodes that lead to even lower levels, creating a hierarchy allowing us to describe structures such as nested loops. With this approach the tool has unique dataflow graphs that exist in different contexts and is capable of handling them accordingly, without developing highly complex algorithms that deal with a single graph with many different exceptions and properties.

3.4.2 Structure of the backend

As for the structure of the backend, at the moment the backend is divided into seven stages as shown in Figure 3.4. These stages are: Graph Initializations, Output Analysis, Parallel Matching, Sequential Matching, Data flow Optimizations, Graph Unfolding and C code generation. From herein the stages are identified by the order in which they are applied (see Figure 3.4).

The approach of the framework is to initially prune the DFG and compact it. Afterwards, the tool optimizes the execution of compacted dataflow. It is important to identify repeating patterns that can be folded. Since the patterns occur multiple times by optimizing their sequence, a large part of the execution is optimized. After the tool compacts the dataflow it can then restructure it. Specifically, the first three stages deal with editing and identifying the loops, the subsequent 3 stages optimize the dataflow, and the last stage produces the C output with possible directives. The

next subsections gives an overview of each stage, and in the next chapter each stage is described in depth while using code examples.

3.4.3 First Stage: Graph Initializations

The first stage is dedicated to some mandatory steps to edit and analyze the initial input DFG prior to the next optimizations. The most essential are the removal of unnecessary nodes, which can vastly improve the execution time and complexity of further algorithms, and the addition of Start and End nodes, which are essential to all algorithms.

3.4.4 Second Stage: Output Analysis

The second stage analyzes the outputs and prepares the graph if necessary for the next stage. The initial graph might be very large and there may exist many different ways to make it more compact. As Parallelism is a property the approach seeks to take advantage of, the start of the folding focuses on parallelization of the generation of the outputs. In case the kernel has a single output this stage as well as the next are skipped. In case of the output being an array the tool separates the dataflows for every individual output value. Then, the tool separates all the common operations from the ones that are unique to every output. The upcoming stage uses these separate dataflows to identify patterns and build loops. Therefore, if they include operations that are common these can be compacted into a loop leading to a far more efficient implementation, since these only need to be executed once in the course of the program, because that single result is used unchanged for the calculation of every output. So after this stage and in case of multiple outputs, the tool produces a graph with all the common operations and then a list of graphs of unique sequences responsible for each output. Before the next stage, the tool needs to actually order the list to simplify the next stages. In case of the outputs being various accesses to the same array, the list gets ordered in the ascending order of the indexes. So if the tool has an output vector $A[n]$ the list follows the order $A[0], A[1], \dots, A[n-1]$.

3.4.5 Third Stage: Parallel Matching

The third stage compares the sequence of each output. The goal is to determine whether it can combine these in a loop. Of course if there are no parallel outputs this step is skipped. If successful, the tool is able to compact the loop into a node and represent all the iterations with a single DFG. As this merely compacts the input graph, which does not lead to a new improved C output, the tool needs to optimize the dataflow. This stage allows for large compressions of the graph that makes further optimizations more efficient. However, by compressing all the separate iterations into a single loop, the tool decreases ILP. Further stages are dedicated to decompressing these loops.

3.4.6 Fourth Stage: Sequential Matching

The purpose of the fourth stage is to optimize the current version of the graph by implementing a pipeline. In the third stage, the tool dealt with the parallelization of the outputs. However, the tool can still have a large graph with a lot of opportunities to optimize. The tool can continue matching the graph to find other parallel loops, but this can lead to a lot of time dedicated to building non-efficient loops. It can have, for example, multiple assignments in parallel and make a loop out of these. This folding only worsen the dataflow by hiding ILP, and complicates the graph leading to less efficient algorithms. As the related work indicate loop pipelining in hardware is a powerful method to optimize algorithm execution. Thus, in this stage the tool identifies a potential variable under certain criteria and pipelines the graph along this variable. To identify the variable, the tool passes through the dataflow from every input node to the output, and records for every separate pass the amount of times a variable is written to. It then compares the records of every pass and selects the maximum for every variable. Then, it compares these and selects the variables with the highest maximum sequential writes. After this identification, the tool matches the sequences that generate all these variables. If these sequences are similar, they can be folded and pipelined to increase the efficiency of the dataflow. The pipeline can travel across the hierarchy developed in the previous stage. In case the pipeline breaks the previous loop, the tool prioritizes the latter. The criteria is a heuristic, and the reasoning is that without knowledge of the input code it is the most probable way of identifying the largest pipeline. This stage can generate a pipeline identical to the one in the original code or in certain cases generate a better one. The fact that the tool can identify other pipelines than the opportunities present in the original code is important, because it shows that the tool can identify better ways to represent the algorithm for HLS tools.

3.4.7 Fifth Stage: Dataflow Optimizations

The next stage is dedicated to implementing optimizations in the dataflow. Currently, the tools optimizes arithmetic operations and memory accesses. These optimizations can have a large impact on the performance in many different ways. Optimization of memory accesses can lower the amount of memory reads in loops. Memory reads are a significant source of bottlenecks in the execution and by lowering them the performance is improved. Currently, the approach to improving them is through array partition and memory reuse. Another method of optimizing the memory accesses is by applying array partition directives of Vivado HLS to be able to have more concurrent memory accesses in hardware. The tool analyzes the memory accesses and identifies the necessary level of memory partitioning to minimize the memory bottleneck. As for the arithmetic optimizations the tool detects chains of identical commutative operations in a row in the DFG. It improves this chain by restructuring it as a tree which enables more ILP. The tool also optimizes divisions in the DFG. Divisions are costly operation in resources and time for hardware implementations. Therefore, if multiple divisions have a common divisor the tool restructures the DFG to first calculate the inverse of the divisor and then replace the division with multiplications

with the inverse value. Thus, divisions are replaced by multiplications that are more efficient in hardware.

3.4.8 Sixth Stage: Graph Unfolding

The sixth stage unfolds to a user defined extent the loops the tool generated. In the first two steps the input graph is compacted. This action can simplify the optimizations and minimize resource usage, but the tools also needs to unfold the loops to take advantage of ILP, since without this crucial aspect all these optimizations might not have a positive impact. The unfolding graph process depends on the type of loops the tools has generated, since the stage four loops have interdependencies while the stage three loops were completely independent. The reason for the cycle between the sixth and fifth stages is due to the fact that every time a loop is unfolded that changes the graph presenting more opportunities for dataflow optimizations. By allowing controlled unfolding, it is ensured that users have higher control on hardware resource usage to satisfy their design goals. The tool also automatically injects memory partition directives to satisfy the indicated number of load/stores.

3.4.9 Seventh Stage: C code generation

The last stage is dedicated to writing the output C code with directives from the given graph. Before the tool outputs the code it passes through the graph to check all the variables in use so that they are correctly initialized. This has to be done at the end since the many variables could be removed or inserted. The tool also levels the graph so that it can identify the order in which the operations are to be written.

3.5 Summary

In this chapter we presented the implementation of the framework. The task of the frontend of the approach is to generate a DFG representing an execution trace of a program. The first section details the way the DFG should be structured to represent various operations and the information it should contain. The second section explained how to insert the appropriate instrumentation code to generate the trace from an execution. The next sections detailed the implementation of the backend. The first presented an overview of its purpose and the next presented new node types introduced in the backend. The final section detailed the structure of the backend that is composed in seven stages that are introduced in the subsections. The stages were briefly described as next chapter explains them in detail.

The current frontend of our approach is still in an initial stage. It does not deal with all forms of potential C constructs like multi-input function calls. There are also certain constructs, such as conditional statements, that would require the approach to be reevaluated to settle on a clear approach to handle these constructs. Additionally, the injection of the instrumentation code is still done manually instead of automatically. However, the presented approach is already capable of

representing many algorithms that do not use such constructs. These algorithms can have many typical C constructs such as array accesses. The approach is clear and can be easily applied to different codes. When the corrected instrumentation code is added the resulting DFG is an accurate representation of the algorithm, in the form that is required for the the backend of the tool. As for the backend it is capable of restructuring the input DFG completely automatically. The backend is more developed than the frontend, as it consists of a more versatile structure capable of applying many different optimizations. The backend can optimize different aspects of the DFG such as its structure or the memory usage. Due to all the optimizations the resulting code can be very different from the original version. Users can guide the backend optimizations through some configurations that are easy to apply. Despite its more developed state, the backend is also in a initial stage and there are still many aspects of the backend to improve.

Chapter 4

Backend C Code Restructuring

This chapter presents a detailed description of the implementation of all the stages of the backend of the framework. To help describing the stages, the *filter subband* benchmark is used as an example. Figure 4.1 shows the graph for two outputs. The dataset size is small to maintain visual clarity. Listing 4.1 shows the original C code for the algorithm.

```
void filter_subband_double_golden(double z[Nz],
double s[Ns], double m[Nm]) {
    double y[Ny];
    int i, j;
    for (i=0; i<Ny; i++) {
        y[i] = 0.0;
        for (j=0; j<(int)Nz/Ny; j++)
            y[i] += z[i+Ny*j];
    }

    for (i=0; i<Ns; i++) {
        s[i]=0.0;
        for (j=0; j<Ny; j++)
            s[i] += m[Ns*i+j] * y[j];
    }
}
```

Listing 4.1: *filter subband* original source code

4.1 Initializations

Upon Loading of the input graph, the tool adds the Start and End nodes to define the dataflow. These nodes are essentials as they are the starting and end points of multiple algorithms. All non "nop" nodes in the graph without entering edges are connected to the Start node and the nodes without outgoing edges are connected to the End node. Afterwards, the tool passes through the graph to gather information for the eventual output C code. It records all the different variable

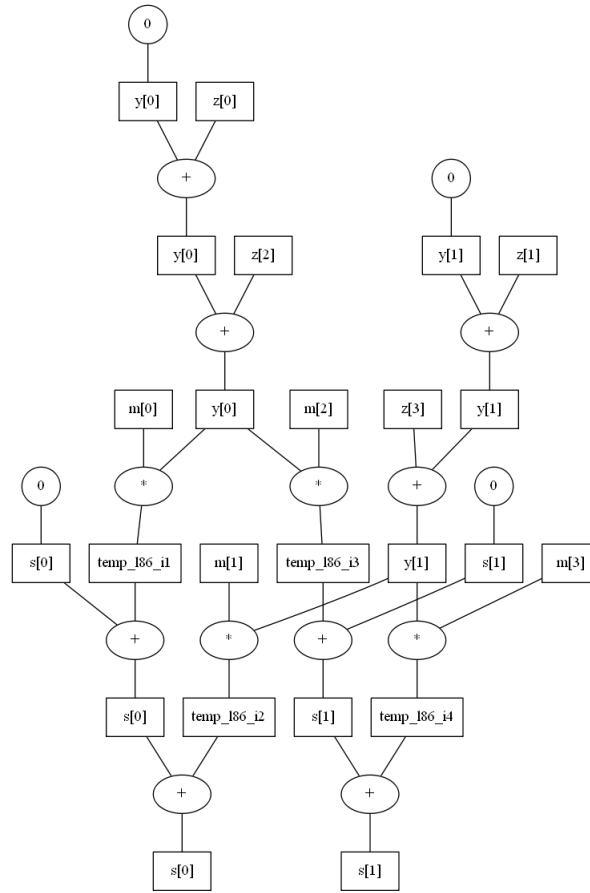


Figure 4.1: DFG of the *filter_subband* benchmark considering an execution with Nz, Ns, Nm and Ny equal to 4, 2, 1024, and 2, respectively

names and types as well as size and dimensions of arrays. After these initializations the tool compacts the graph by pruning nonessential nodes. These are variable nodes that exist between operations. They were essential in generating and connecting the initial graph at the frontend, but for the analysis they are not relevant. This information can be more efficiently stored on edges between operation nodes since for the dataflow analysis the most essential aspect is the sequence of operations. It also leads to simpler representations of operations, since now they are all defined by a single node instead of a connection between multiple nodes of different types. The tool also changes nodes to represent assignments. The variable nodes at the start and end stay unchanged since they represent the dataflow interface. The pseudo code in Algorithm 1 details the approach. This optimization greatly reduces the number of nodes in the graph leading to faster and simpler algorithms. After pruning the algorithm advances to the next stage. Figure 4.2 shows the result of the Stage 1 steps for the *filter_subband* benchmark. When compared to the DFG prior to initializations (seen in Figure 4.1) the differences are clear. The beginning and end of the dataflow is defined by specific nodes. Between the inputs and outputs there are only operation nodes with the variables being indicated in the edges between the operations. Additionally the graph shows that the array *y* has been replaced by variables such as *y0*. The new representation is then inputted

to the next stage.

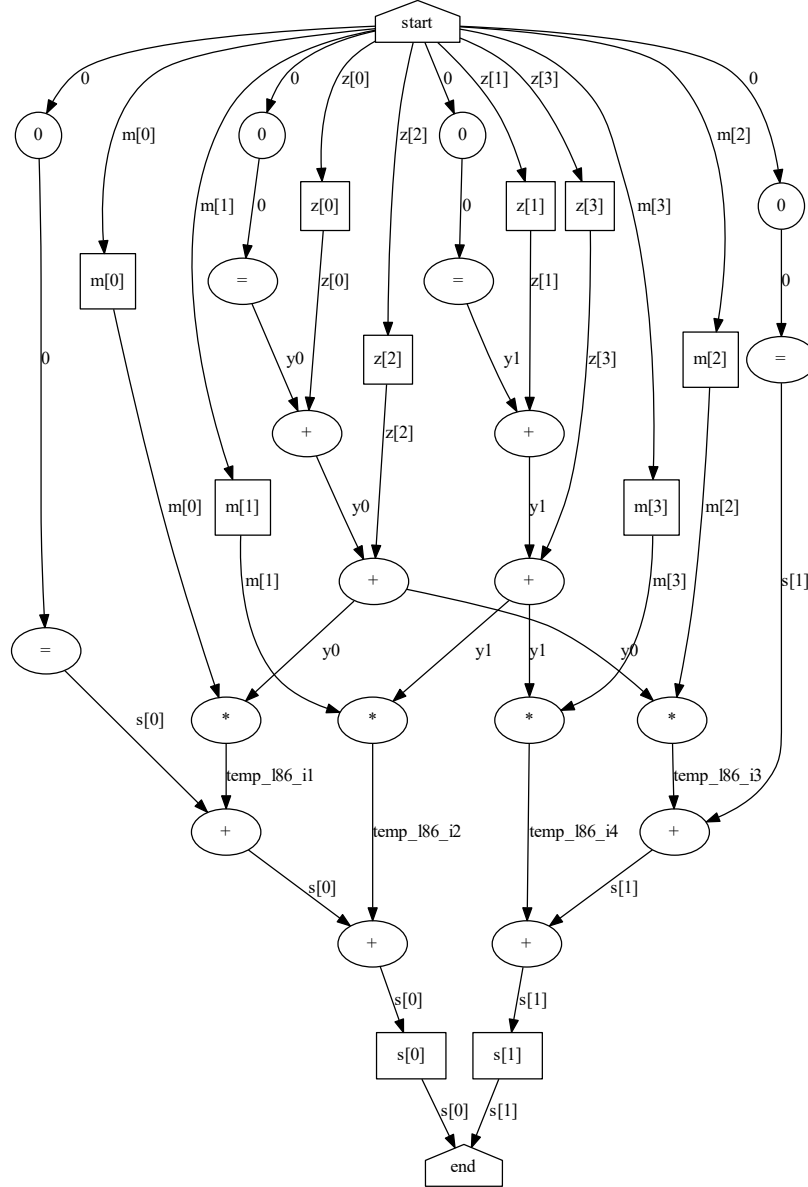


Figure 4.2: *filter subband* after stage 1 initializations

4.2 Output Analysis

The output analysis stage is dedicated to analyzing the graph for further analysis. If the original code consists of loops without ifs then the dataflow consists of multiple similar sequences that generate different outputs. With this knowledge the tool attempts to compact the subgraphs by identifying parallel outputs' dataflows that can be looped. This stage prepares the tool for the

Algorithm 1 Intermediate node removal

```

1: procedure INTERMEDIATE NODE REMOVAL(Graph G(V,E))
2:   for each node  $n$  do
3:     if  $n.type = var$  then
4:        $p \leftarrow pred(n)$ 
5:       if  $p.type = op$  AND  $succ(n) = op$  then
6:         for every node  $n2 \in succ(n)$  do
7:            $G.add\_edge(p, n2)$ 
8:            $copy\_attributes(n, edge(p, n2))$ 
9:         end for
10:         $G.node\_remove(n)$ 
11:      else if  $p.type = var$  then ▷ specific case of assignment
12:         $copy\_attributes(p, edge(p, n))$ 
13:        for every node  $n2 \in succ(n)$  do
14:           $copy\_attributes(n, edge(n, n2))$ 
15:        end for
16:         $clear\_attributes(n)$ 
17:         $set\_nodetype(n, op)$ 
18:         $set\_nodelabel(n, =)$ 
19:      end if
20:    end if
21:  end for
22:  return G
23: end procedure

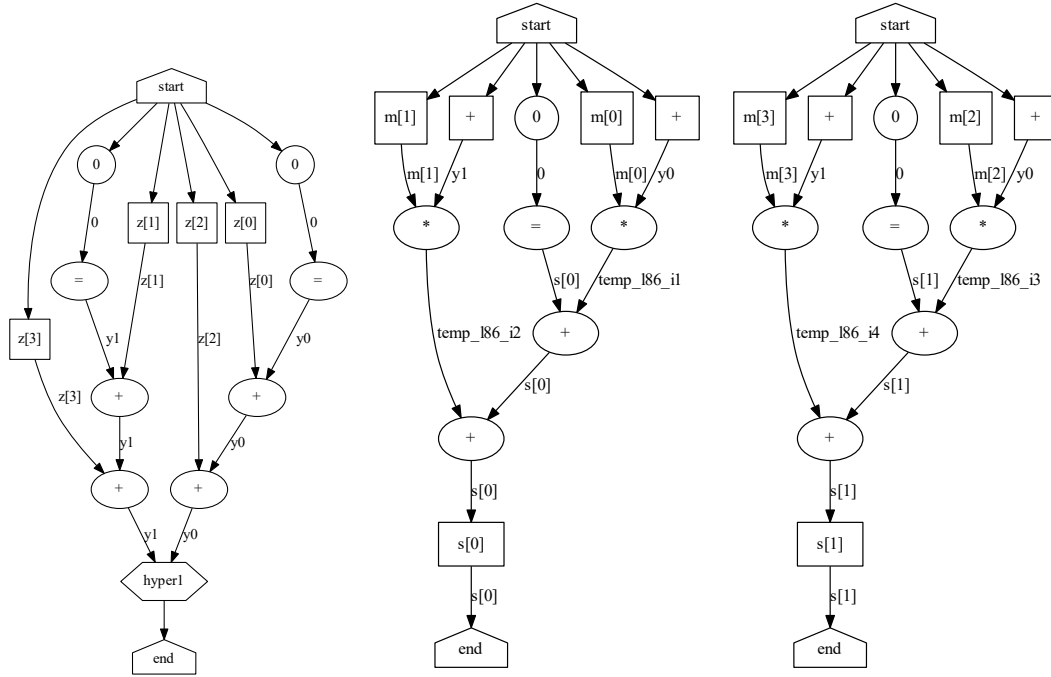
```

matching in the next stage. The first step is to identify the outputs. In case the output is a single variable the tool can skip this stage and the next. In case the output is an array the tool decomposes the dataflow that lead to each output. The tool applies the Algorithm 2 to acquire a list of dataflows. Each member of the list is a subgraph that generates a single unique output value. The tool identifies these sequences by starting at the endpoint of each subgraph and moving up in the subgraph. Every node and edge encountered are added to a new graph.

The sequences created in this ways might still not be independent since they can share operations that are in fact done only once, and whose result is used for every output. It is important then to separate these operations, since they would lead to generating loops with unnecessary instructions, slowing down the execution severely. The problem is handled by Algorithm 4. It identifies common operations by comparing the ids of nodes. As mentioned in Chapter 3 any new node has a unique id. When the tool creates the new graphs it reuses these ids. So, if there are matching ids between subgraphs, then that exact operation was executed in both sequences. If this id is used in all sequences the operation is common to all sequences. These nodes are removed from the separate sequences and placed in a graph dedicated to common nodes. The only common nodes that are not removed from the separate graphs are the ones with an outgoing edge that is used for a non-common operation. If a node has this quality it means that it is on the border between common and non-common nodes and it has to be kept so that algorithms can correctly pass between hierarchies. Without them the tool would not be capable of knowing how to continue when passing through hierarchies. This separation leads to an upper graph with all the operation that are executed for every output. This graph has a hyper node that then contains the list of all individual sequences.

To avoid comparing every node of a sequence with all the nodes of the other sequences and thereby having exponential complexity the tool first levels the graph. Leveling is implemented by the algorithm described by the pseudo code Algorithm 3. The result is a list of levels each containing nodes. The levels indicate when an operation can be executed. If an operation has no predecessor it can execute immediately and is placed at the first level. If operation has predecessor it is only placed at a level after all its predecessors are leveled. In that case it is placed on the level that comes after the highest level of its predecessors. The reason for the leveling is that if the dataflows have equal operations in each sequence, then they will execute at the same level. So, for the separation algorithm the tool only compares nodes of the same level thereby lowering the complexity. After obtaining the list of separate dataflows, the tool orders them for the next stage. If the output is a single dimensional array the outputs are ordered in ascending value of the index. So for a array $x[n]$ the tool generates the order $x[0], x[1], \dots, x[n]$. In case of multiple dimensions the tool orders them by ascending order and from the first dimension to the last. So, array $m[n][n]$ gets ordered as $m[0][0], m[0][1], \dots, m[0][n], m[1][0], \dots, m[1][n], \dots, m[n][n]$.

Figure 4.3 shows the result of this separation for two outputs. Compared to Figure 4.2 there is a clear separation of the dataflow. The common subgraph that exist on the upper level (see Figure 4.3a) is the dataflow that calculates the y values of the kernel. These exact values are used in the calculations of both of the outputs as the code in Listing 4.1 and the original dataflow in Figure



(a) Common subgraph of the graph (b) Unique subgraph for first output (c) Unique subgraph for second output

Figure 4.3: *Filter subband* benchmark after second stage of the tool and considering an execution with Nz, Ns, Nm and Ny equal to 4, 2, 1024, and 2, respectively

4.2 indicate. After the calculation of the y values then there is no more dependences between the dataflows that generate the outputs. Then these flows are separated into their individual sequences as shown in figures 4.3b and 4.3c. In those sequences, some common nodes are kept, such as the nodes that produce the definitive y values. If the tool passes from the upper level to one of the subgraphs of lower level by going from the node that produces $y1$. It knows that the lower level graph will have an equivalent node with the same identifier so it starts the algorithm directly from that equivalent node.

In every graph the first level of nodes are always input nodes that connect to the interface of the dataflow of other levels, so the tool handles them differently. For example even if they are of the type "op" the writing Algorithm 10 never views them as valid operations and therefore does not write the C code that they represent. Even though the dataflow repeats the operation nodes, the actual statement of the operation appears only once in the output C code at the correct position.

Algorithm 2 Get Individual Output Sequence

```

1: procedure DETECT INDIVIDUAL OUTPUT SEQUENCE(Graph  $G(V,E)$ )
2:   graphlist=new_List()
3:   for each node  $n$  pred( $G.End$ ) do
4:      $G' \leftarrow$  new_graph()
5:      $G'.addnode(n)$ 
6:     nodelist.add( $n$ )
7:     repeat
8:        $n' \leftarrow$  nodelist.head()
9:       for every node  $p \in$  pred( $n'$ ) do
10:        if  $p \neq G.Start$  then
11:           $G'.addnode(p)$ 
12:          nodelist.add( $p$ )
13:        end if
14:      end for
15:      nodelist.remove( $n'$ )
16:    until nodelist is empty
17:    graphlist.add( $G'$ )
18:  end for
19:  return graphlist
20: end procedure

```

Algorithm 3 Leveling Algorithm

```

1: procedure LEVELING ALGORITHM(Graph G(V,E))
2:    $level \leftarrow 0$ 
3:   current_level.add(G.Start)
4:   Start.add_attribute(level)
5:   list_of_levels.add(current_level)
6:   current_level.clear()
7:   for each node  $n \in \text{succ}(G.Start)$  do
8:     nodelist.add(n)
9:   end for
10:  repeat
11:     $level \leftarrow level + 1$ 
12:    for every node  $n' \in \text{nodelist}$  do
13:      if pred( $n'$ ) is leveled then
14:        current_level.add( $n'$ )
15:        mark  $n'$  as leveled
16:      end if
17:    end for
18:    list_of_levels.add(current_level)
19:    nodelist  $\leftarrow \text{nodelist} \ominus \text{current\_level}$ 
20:    for every node  $n'' \in \text{current\_level}$  do
21:      nodelist.add(succn( $n''$ ))
22:    end for
23:    current_level.clear()
24:  until nodelist is empty
25:  return list_of_levels
26: end procedure

```

▷ remove leveled nodes from list
 ▷ add successors to list

Algorithm 4 Separation Algorithm

```

1: procedure SEPARATION ALGORITHM(GI)
2:   Commongraph  $\leftarrow$  new_graph()
3:   leveledgraphlist  $\leftarrow$  Leveling Algorithm(GI)
4:   for every level le  $\in$  leveledgraphlist do
5:     for every node n  $\in$  le  $\in$  leveledgraphlist.head() do
6:       for every node n'  $\in$  le do
7:         if n.Id = n'.Id then
8:           matches  $\leftarrow$  matches + 1
9:         end if
10:      end for
11:      if matches = graphlist.size then
12:        commongraph.addNode(n)
13:        connect_node(n)
14:        n.addAttribute(common,true)  $\triangleright$  every attribute has a label and a value
15:      end if
16:    end for
17:  end for
18:  for each Graph G  $\in$  GI do
19:    for each node n  $\in$  G do
20:      if n.hasAttribute(common) AND succ(n).hasAttribute(common) then
21:        graphlist.remove_node(n)
22:      end if
23:    end for
24:  end for
25:  commongraph.addNode(hyper)  $\triangleright$  hyper node that holds the graphlist
26:  hyper.addAttribute(GI)  $\triangleright$  place the separate sequences on another level
27:  return commongraph
28: end procedure

```

4.3 Parallel Matching

In this stage the tool handles the matching of the individual sequences represented as subgraphs that the tool creates in the previous section. The goal is to identify a single dataflow that can be written in a loop to represent all parallel dataflows. In the graph two nodes matching means that the nodes are the same operation and the edges are reproducible. Edges being reproducible in this context depend on the type of edge. If its a constant then they have to be the same value. If it is a non array variable they only need to be of the same type. Since in the DFG all the edges are connections of values the name of the variable is not important for the matching. For vectors the edges need to have the same name and a consistent way of calculating the next index. For example, Figure 4.4 shows the dataflow for four outputs of the *filter subband* benchmark. The purple node of the dataflow in Figure 4.4a can reproduce the purple and blue nodes of the dataflow in Figure 4.4b, since the variables are of the same type, the vector access are of the same vector and $m[0]$ can reproduce $m[4]$ of the purple node by merely adding 4 to the index or $m[5]$ of the blue node by adding 5 to the index. Once these nodes are matched to the dataflows in figures 4.4c and 4.4d then the only viable match is the purple nodes, since $m[8]$ can be accessed by adding 4 to to the index $m[4]$ and $m[12]$ by adding 4 to the index of $m[8]$. Other matches are not possible since there are no ways of having a consistent method of calculating the next index. For example matching the red node with blue node and the green node of the dataflows in figures 4.4a, 4.4b and 4.4c respectively, is impossible since to reach the blue node from the red node the index should be incremented by 2. But to reach the green node from the blue node the index need to be incremented by 5. Without using conditional statements an index cannot increment by two in one iteration and five in another, therefore the matching is not possible.

This notion is expanded for entire sequences. Sequences are reproducible if every single node in a sequence has a matching one in the other, and if two nodes match then the parent nodes must also match. For example, figures 4.4a and 4.4b show that any nodes of the first dataflow, that represent an addition, can be matched with any other node that represent an addition of the other dataflows . If the pink node of the second dataflow is compared with the light blue of the first they match. Once the parent nodes are compared these will not match, since they are different types of nodes. Therefore there is no matching sequence containing that specific match. If the matching process is started with the light blue nodes of the first two dataflows then it is possible to match the two dataflows up to the inputs. Finally, to have a loop it must be ensured that the inputs and outputs between the matched sequences are similar. While matching the body of the loop a name of a variable is not significant but if that variable is used outside of the loop it is important to ensure the connection is correct.

Thanks to the preparation in the previous stage, the complexity of this section is lower. If the tool made no assumptions then it would need to compare every node with every other node of other sequences. This easily leads to an extremely large number of possible comparison, with many potential non optimal paths or failed paths. Also blindly matching can develop multiple small loops and HLS tools can have difficulties exploiting ILP between sequences of loops, leading to

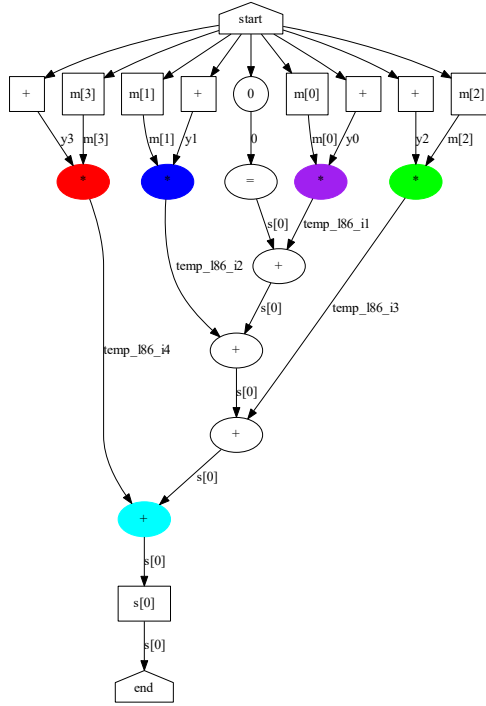
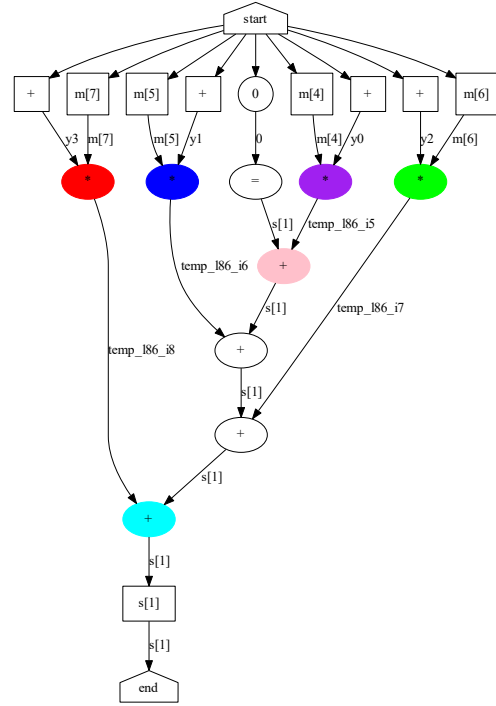
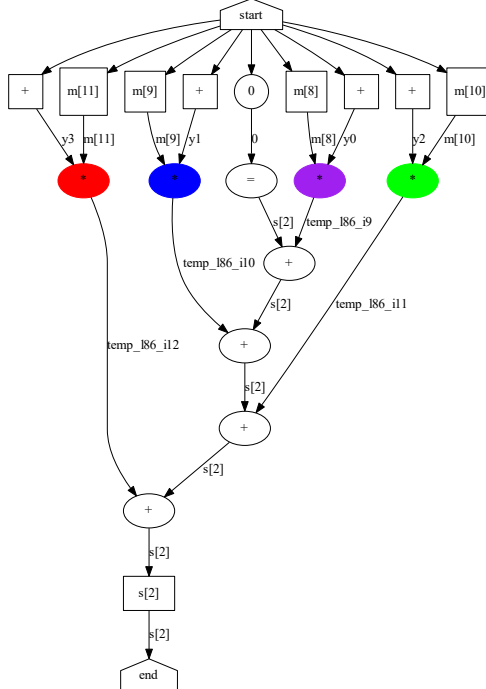
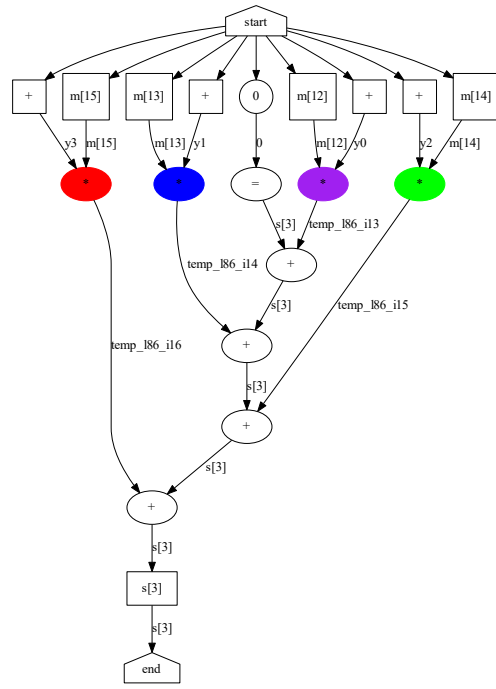
(a) Dataflow that calculates $s[0]$ (b) Dataflow that calculates $s[1]$ (c) Dataflow that calculates $s[2]$ (d) Dataflow that calculates $s[3]$

Figure 4.4: Separate sequences for *filter subband* considering an execution with N_z , N_s , N_m and N_y equal to 8, 4, 1024, and 4, respectively

non optimal hardware implementations. Even in the previous examples many different matches were shown. Thus, the tool simplifies the problem due to the preparation and by specifically aiming to match the output generation.

The tool applies the matching algorithm described in Algorithm 6. It starts from the bottom which are the output nodes. The tool creates a list of nodes to compare for every DFG. The lists are initialized with the bottom output nodes. The tool then compares the nodes on the list of a graph with only the nodes on the list of the next graph. This is because the tool needs to check if the nodes can be reproduced by the previous iteration. The nodes are compared through the labels and the edges are compared using Algorithm 5. In this algorithm the tool has an exception for when both of the edges are not checked. If the first one has been checked the tool only needs to mark the next one. This is especially important for vector access because in the first comparison the tool does not demand a specific increment between iterations. However, the next comparisons need to have the same increments as the first two since then the tool could not generate a loop. The tool always compares input edges taking into account whether they are right or left hand operands. If a comparison is successful the tool adds their parent nodes to the list of nodes to compare. The tool always adds them in order of right and left operand so that the list stays ordered. This means that the tool always knows exactly which parent nodes to compare thereby lowering the amount of unnecessary comparisons. After a node has passed all comparisons the tool adds it to the new graph that is going to represent the loop.

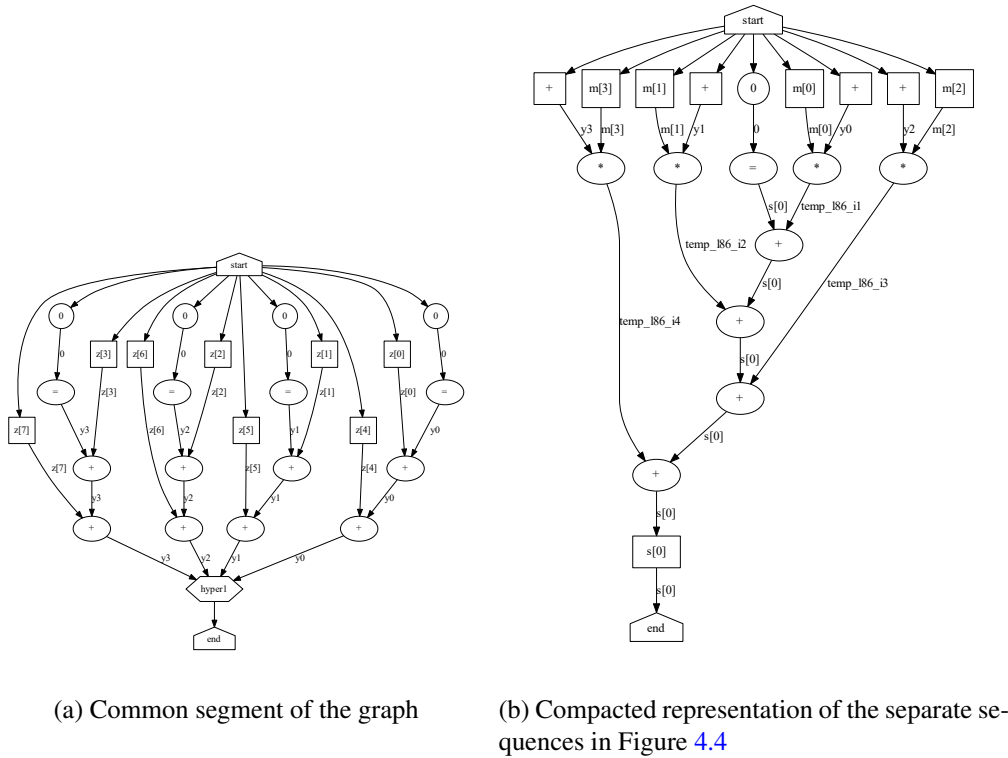


Figure 4.5: *Filter subband* after second stage of the tool and considering an execution with Nz, Ns, Nm and Ny equal to 8, 4, 1024, and 4, respectively

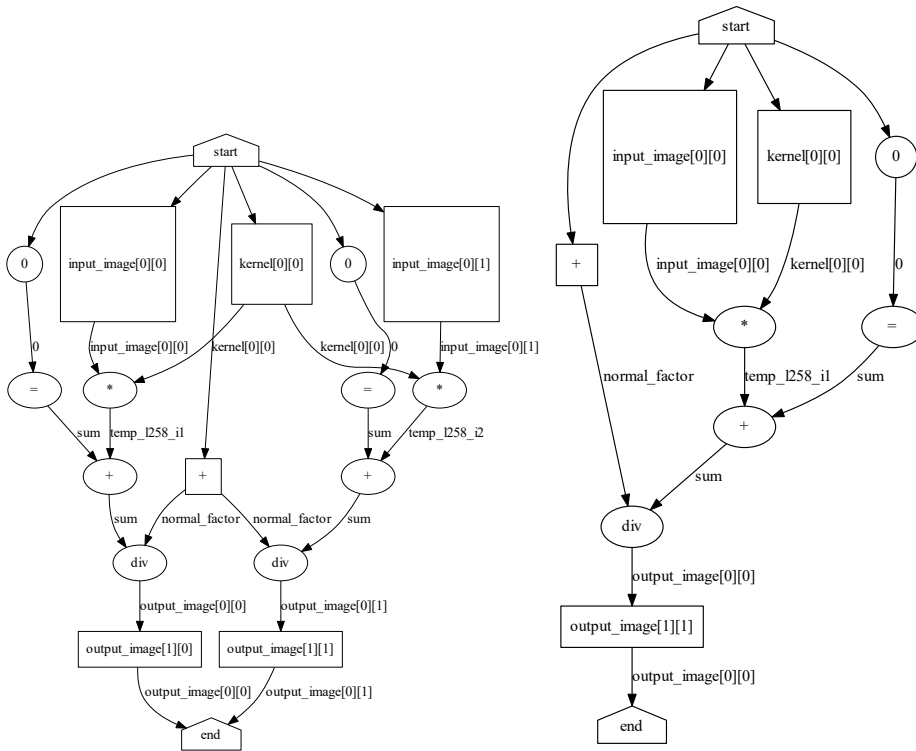
The tool bases the compact dataflow on the lowest indexed dataflow, since it is going to be the first iteration of the loop. Therefore, the edges have the same labels as the first dataflow. Edges corresponding to vector accesses also have information on how much to increment for the next iteration of the loop. In case, there is a mismatch the tool has to remove the mismatched paths from the comparisons list. The tool saves the nodes that cause the mismatch in a list. And after generating the loop it places the mismatched dataflows outside the loop.

Figure 4.5 shows the matched nodes and the resulting sequence. In case of success the tool is able to represent multiple sequences by a single one. Listing 4.2 shows the equivalent C code. This code now produces the outputs in a loop. That loop contains a pipeline directive since loops with independent iterations are improved through pipelining. Comparing this code with the original version we conclude that the difference is the pipeline directive, the *y* vector and that all the loops except the outer loop of second loop are completely unfolded. With the exception of the implementation of the *y* vector as variables lowering the amount of BRAMs, there has been no significant code restructuring that could not have been implemented by basic directives. However, a more compact representation of the dataflow is already obtained. This aspect is crucial for further optimizations, since changing the loop dataflow optimizes all the iterations of that loop. Therefore the complexity and time investment is lower. The code also shows why the separation of the previous section is crucial. In this case the calculation of the *y* values is common to all the outputs. If they were kept in the unique sequence then they would have been successfully matched and included in the loop, because their pattern is exactly the same for every output. So the loop would calculate every *y* value per iteration, which is completely inefficient and would generate a far worse implementation. The next stages apply larger restructuring to the graph that lead to the more complex optimizations and better results.

```
void filter_subband_double_golden(double z[Nz],
double s[Ns], double m[Nm]) {
    double y[Ny];
    y0=0;
    y0=y0+z[0];
    ... \\calculation of y values
    y3=y3+z[7]
    for (i=0;i<4;i++){
        #pragma HLS pipeline
        s[i]=0;
        temp_186_i1=m[i*4]*y0;
        temp_186_i2=m[i*4+1]*y1;
        ...
        s[i]=s[i]+temp_186_i1;
        ...
        s[i]=s[i]+temp_186_i4;}
}
```

Listing 4.2: *Filter subband* equivalent output code after stage 3 folding

The tool is also capable of generating nested loops, e.g. to compact two dimensional arrays. In the C language to pass through all elements of a multi dimensional array nested loops can be used. To generate nested loops the tool simply applies the matching algorithm multiple times. Every pass compacts the subgraphs along one of the dimensions of the array. Stage 2 aligned the values so that the first pass compacts the loop along the first dimension, the second pass along the second dimension. The benchmark *2D Convolution* that filters a 2D image by a 2D kernel is an example of the 2D folding. Figure 4.6 shows the dataflows of the inner loops after the first and second pass through the matching algorithm. Looking at both dataflows, it is clear that the one in Figure 4.6b is a folded version of the one in Figure 4.6a.



(a) dataflow of the inner loop after first pass (b) dataflow of the inner loop after second pass

Figure 4.6: Inner loops of the *2D Convolution* benchmark during stage 3 four considering an execution with N, K, equal to 2 and 1, respectively

```
void conv2d_first(int input_image[2][2],
int output_image[2][2], int kernel[1][1]){

    \\calc of normal factor
    for (i=0;i<2;i++){
        #pragma HLS pipeline
        sum1=0;
        sum2=0;
```

```

    temp_l258_i1=input_image[i][0]*kernel[0][0];
    temp_l258_i2=input_image[i][1]*kernel[0][0];
    sum1=temp_l258_i1 +sum1;
    sum2=temp_l258_i2 +sum2;
    output_image[i][0] = sum1/normal_factor;
    output_image[i][1] = sum1/normal_factor;
}
}

void conv2d_second(int input_image[2][2],
int output_image[2][2], int kernel[1][1]){

    \\calc of normal factor
    for (i=0;i<2;i++){
        for (j=0;j<2;j++){
            #pragma HLS pipeline
            sum=0;
            temp_l258_i1=input_image[i][j]*kernel[0][0];
            sum=temp_l258_i1 +sum;
            output_image[i][j] = sum1/normal_factor;
        }
    }
}
}

```

Listing 4.3: 2D Convolution stage 3 output

4.4 Sequential matching

In the previous step, the tool attempts to compact sequences known as parallel. After compacting the outputs, the tool obtains a more compact dataflow that can generate every output. This can still be very large and contain properties left to explore. In the fourth stage the tool identifies a potential variable satisfying certain criteria and pipelines the graph along this variable. The tool applies the pipelining at Stage 4 to DFGs that generate a single output. Thus, either the input graph only has a single output or the input DFG was successfully compacted in Stage 3. The tool chooses the variable which is written more times sequentially. The tool determines this variable by passing through every input down to the output. For every pass it stores how many times a variable was written. The tool compares the results for every input. The variable most often written is then identified and selected for the optimization. The tool sets in a list all the writes of the chosen variable and orders them in the order of first to last write.

The tool matches the dataflows that generate all these different values of the variable using an algorithm similar to Algorithm 6. A substantial difference from the previous algorithm is that the pipeline matching can traverse the hierarchy upwards. It starts at the lowest hierarchy in which the variables were written to and after completing the matching at that level it can pass to the upper

Algorithm 5 Compare edges

```

1: procedure COMPARE EDGES(Graph G, Edge E, Edge E')
2:   if E.type = const AND E'.type = const then
3:     if E.name = E'.name then
4:       if E.has_mark then
5:         E'.addAttribute(mark, true)
6:       return true
7:     else
8:       MARKFIRSTEDGES(E, E')
9:       return true
10:    end if
11:  end if
12:  else if E.type = var AND E'.type = var then
13:    if E.is_array = false AND E'.is_array = false then
14:      if E.var_type = E'.var_type then
15:        if E.has_mark then
16:          E'.addAttribute(mark, true)
17:        return true
18:      else
19:        MARKFIRSTEDGES(E, E')
20:        return true
21:      end if
22:    end if
23:  else
24:    incr ← E'.index - E.index
25:    if E.has_mark then
26:      if E.name = E'.name AND increment = G.get_edge(E).increment then
27:        E'.addAttribute(mark, true)
28:      return true
29:    end if
30:    else if E.name = E'.name then
31:      MARKFIRSTEDGES(E, E')
32:      G.Edge(E).addAttribute(increment, incr)
33:    return true
34:  end if
35: end if
36: end if
37: return false
38: end procedure
39: procedure MARKFIRSTEDGES(Edge E, Edge E')
40:   G.addNode(pred(E))           ▷ A node has to be added to add the matched edge
41:   E.addAttribute(mark)
42:   E'.addAttribute(mark)
43: end procedure

```

Algorithm 6 Parallel Matching

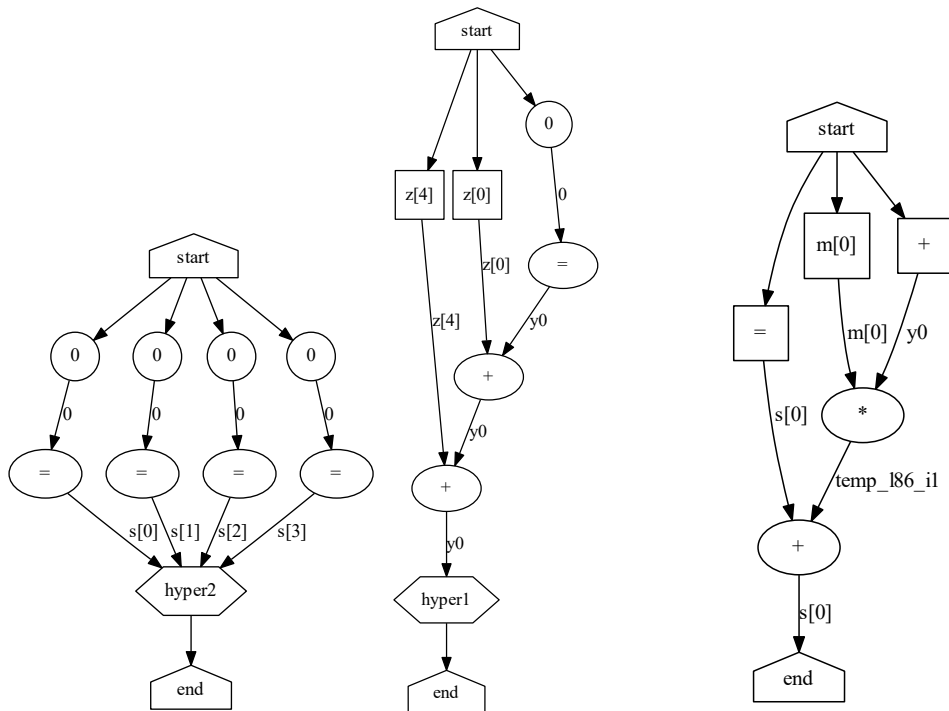
```

1: procedure PARALLEL MATCHING(graphlist G1, G)
2:   for every Node End  $\in$  G1 do
3:     nodelist.add(pred(End))            $\triangleright$  start algorithm with bottom node of every graph
4:     nodelist_l2.add(nodelist)          $\triangleright$  nodelist_l2 is a list of nodelist
5:   end for
6:   repeat
7:     for  $i \leftarrow 0 : \text{nodelist\_l2.size} - 1$  do
8:       mismatch  $\leftarrow$  false
9:       for  $j \leftarrow 0 : \text{nodelist.size}$  do
10:        Node n  $\leftarrow$  nodelist_l2.get(i).get(j)
11:        Node n'  $\leftarrow$  nodelist_l2.get(i+1).get(j)
12:        check_edges  $\leftarrow$  check_edges AND compare edges(G,n.left,n'.left)
13:        check_edges  $\leftarrow$  check_edges AND compare edges(G,n.right,n'.right)
14:        check_edges  $\leftarrow$  check_edges AND compare edges(G,n.result,n'.result)
15:        if n.name = n'.name AND check_edges = true then
16:          if i=0 then
17:            next_nodelist_l2.get(i).add(pred(n))    $\triangleright$  first add left then right input
18:          end if
19:          next_nodelist_l2.get(i+1).add(pred(n'))
20:        else
21:          mismatch  $\leftarrow$  true
22:          exit loop 9
23:        end if
24:      end for
25:      if mismatch then
26:        exit loop 7
27:      end if
28:    end for
29:    if mismatch then            $\triangleright$  mismatch therefore no longer necessary to compare
30:      if  $i > \text{nodelist\_l2.size} / 2$  then
31:        for  $k \leftarrow \text{nodelist\_l2.size}, i + 1$  do
32:          REMOVE_MISMATCH(k)
33:        end for
34:      else
35:        for  $k \leftarrow 0, i$  do
36:          REMOVE_MISMATCH(k)
37:        end for
38:      end if
39:    end if
40:    nodelist_l2.clear()
41:    nodelist_l2  $\leftarrow$  next_nodelist_l2
42:    next_nodelist_l2.clear()
43:  until nodelist_l2.get(0) is empty
44:  resolve_conflict(conflict)
45:  store loop information in G
46:  return true
47: end procedure
48: procedure REMOVE_MISMATCH(k)
49:   conflict.add(nodelist_l2(k))            $\triangleright$  save conflict nodes
50:   nodelist_l2.remove(k)                  $\triangleright$  remove sequences from matchlist due to mismatch
51:   next_nodelist_l2.remove(k)
52: end procedure

```

level of the graph. The matching ends at the level that the algorithm was called at. If the tool calls it from the upper level then the algorithm stops matching at that level. Due to this characteristic, the pipeline loop is prioritized over the loop generated at Stage 3 and is capable of changing it. The tool normally calls the algorithm from the highest level, except when the highest level has no actual operations, since that would automatically match and destroy unnecessarily the loop created in Stage 3. After the tool obtains the pipeline structure it has to handle the nodes outside the loop. Like in the previous stage, the tool places the graph that represents the dataflow of the pipeline in a hyper node and all the nodes that do not fit in the pipeline are placed outside of the "hyper" node.

The tool handles the nodes depending of the context. If the pipeline does not break the previous loop, the conflicting nodes stay as part of the previous loop. If the previous loop is broken, then the nodes have to be placed at the upper level. That level does not handle compacted nodes so if the conflicting nodes were folded representations of the previous loop, they have to be placed unfolded in the upper level. It is important to place the conflicting nodes at the right positions. In the previous stage all the dataflows were parallel, but this stage applies a sequential fold, therefore it is important to know if the conflicting nodes are to be placed after or before the loop. The configurations allow the user to chose how much to fold. If the user choses medium it only applies the stage 3 folding. If he/she chooses high the tool applies the full folding up to stage 4.



(a) DFG of the highest hierarchy (b) Compact DFG of outer loop (c) Compact DFG of inner loop level

Figure 4.7: *Filter subband* after fourth stage of the tool and considering an execution with Nz, Ns, Nm and Ny equal to 8, 4, 1024, and 4, respectively

The *filter subband* benchmark is a good example of the pipelining algorithm. By analyzing the original code in Listing 4.1 it is clear that every time a y value is calculated it can be immediately used, while other y values are being calculated. This relationship is not explicit in the original code and therefore HLS tools might have difficulties recognizing this opportunity for parallelism. However, through the DFG representation, this parallelism is easier to verify. When the graph of this benchmark reaches stage four, the tool selects to pipeline along the array s , leading to the code in Listing 4.4. This modified description of the algorithm exposes the aforementioned parallelism clearly and implements a pipeline that can take advantage of it. Figure 4.7 shows the dataflow of the pipeline. In Figure 4.7a the operations that do not fit the pipeline are shown. These are the initializations of the s vector. Figures 4.7b and 4.7c display the pipeline dataflow and they are compacted dataflows of the ones in Figures 4.5a and 4.5b.

```
void filter_subband_pipe(double z[8],
double s[4], double m[1024]){
    ... // initilizations of s vector
    for( int i =0; i < 4; i=i+1){
        #pragma HLS pipeline
        y0_w7=0;
        y0_w6=z[i] + y0_w7;
        y0_=z[i+4] + y0_w6;
        for( int j =0; j < 4; j=j+1){
            temp_l86_i1=m[(4)*j+i] * y0;
            s[j]=s[j] + temp_l86_i1;
        }
    }
}
```

Listing 4.4: *filter subband* code output after the pipeline of the fourth stage of the tool

4.5 Graph Unfolding

The sixth stage is dedicated to unfolding loops that were generated in Stages 3 and 4. The sixth stage is presented before the fifth because as the backend structure indicates after every pass through this stage the tool returns to the previous one. Unfolding loops opens new avenues for optimizations in Stage 5, so it is preferable to explain Stage 6 first. As mentioned before compacting the graph is very efficient for optimizations, but to take further advantage of ILP the tool needs to unfold the loops. Algorithm 7 describes the approach taken to unfold a loop. The tool can unfold the loop simply by copying the dataflow multiple times and updating the indexes of vector accesses and appending a label to the new variables. This name change is important to differentiate the new variables and it is only skipped for edges that are inputs and outputs of the loop to maintain correct communication with the outside of the loop. In case of unfolding the pipeline, the tool takes an extra step to connect the copied dataflows according to dependencies. The tool knows which edges to connect, since the tool annotated the connecting edge in the previous stages.

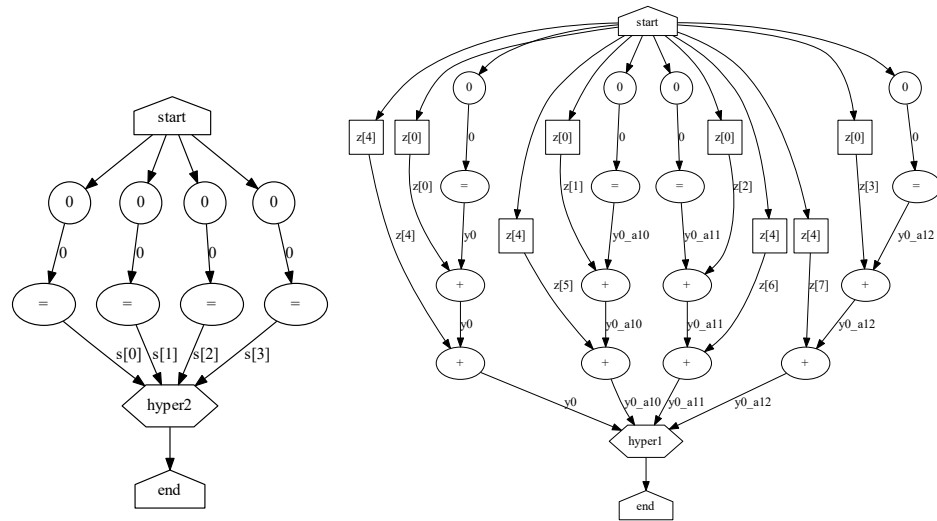
The unfolding process starts at the innermost loop. After a loop is unrolled the resulting dataflow is checked for Stage 5 optimizations, and after these the graph returns to Stage 6. This process ends when there is no more loops to unfold. The unfolding process needs to be ordered from innermost to outermost loop, because unfolding an inner loop does not affect the outer loop. However, unfolding an outer loop affects its nested loops. The only time an inner loop is not unfolded is when it is inside a loop that is going to be pipelined. That is because the pipeline directive of Vivado HLS automatically unrolls all loops inside it.

In the case of the *filter subband* pipeline (see Listing 4.4), the inner loop that calculates the s values will be fully unrolled by Vivado HLS as the outer loop is pipelined, thus it is unnecessary to unroll the inner loop. Figure 4.8 shows the result of unfolding the *filter subband* pipeline loop showed in Figure 4.8. Figures 4.7a and 4.8a are the same because the unfolding stage does not affect non looped dataflows. Figure 4.8b shows the dataflow of 4.7b replicated four times. Analyzing the replicated dataflows, the replicated variables with different names can be seen, such as $y0$, $y0_{10}$, $y0_{a20}$ and $y0_{a30}$. Without the name changes, the tool would output the operations $y0 = z[4] + y0$, $y0 = z[5] + y0$, $y0 = z[6] + y0$, $y0 = z[7] + y0$ which would lead to the wrong implementation. The indexes are updated based on the loop that is being unrolled and the information stored on the edges during the folding. In this case, the folding recorded that in the next iteration the index would increment by 1, therefore one is added to the index. This is clear by checking Figure 4.7b and comparing the dataflows. The original dataflow accesses $z[0]$ and $z[4]$ while the next accesses $z[1]$ and $z[5]$. This is also consistent with the code in Listing 4.4.

The unrolled inner loop is displayed in Figure 4.8c and it is different compared to the original in 4.7c. When dealing with inner loops it is essential to distinguish along which loop to unfold. Listing 4.4 shows that unrolling the access $m[(4 * j) + i]$ is very different depending on the chosen loop. Unfolding along j the next access is $m[(4 * j) + i + 4]$ and unrolling along i the next access is $m[(4 * j) + i + 1]$. Therefore, the tool starts the unrolling algorithm with the name, folding factor and loop type of the initial loop. If that loop has a nested loop, the tool unrolls it and propagates the name of the outer loop, the folding factor and type as seen in line 22 of Algorithm 7. The inner loop is unrolled based on that inherited information. In this case, the tool unrolls the outer pipeline loop, so the inner loop will be unrolled along i , which is verified by the fact that the dataflow uses the accesses $m[0]$, $m[1]$, $m[2]$ and $m[3]$.

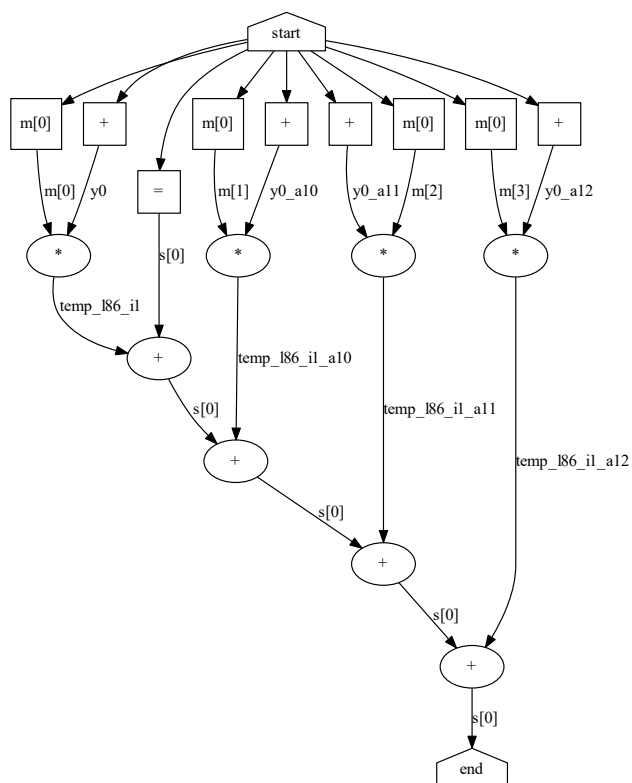
Lastly an important distinction is that these replicated dataflows are not in parallel like in Figure 4.8c. The type of loop the tool is unrolling in this case is sequential, meaning that the iterations are not fully independent. So, if a value from a previous iteration is used in the next iteration the dataflow must represent that. The edges to connect have a unique attribute. In this case, the loop is from Stage 4 and it indicated the edge with label s as the edges to connect between iterations. The Algorithm 7 replicates the dataflows as if they were parallel and then connects them. The result of the connection is the chain in Figure 4.8c.

Algorithm 7 is applied to loops from Stages 3 and 4, but there are some modifications for loops that have been optimized through memory reuse as applied in Stage 5. That optimization changes some properties of the graphs and loops as the next section details. By applying the memory



(a) dataflow of the highest hierarchy level

(b) Unrolled outer loop



(c) Unrolled inner loop

Figure 4.8: Unrolled dataflow of pipelined *filter subband* benchmark of Figure 4.7 by a factor of four considering an execution with Nz, Ns, Nm and Ny equal to 8, 4, 1024, and 4, respectively

optimization the parallel loops are no longer independent, since the memory buffers need to be adequately loaded before the next iteration. Therefore the loops need to be replicated in a manner that expresses that interdependency. The tool replicates them as if the loops were independent, but instead of placing the replicated iteration in parallel the algorithm places them after the dataflow of the previous iteration to maintain the correct execution order. After the dataflows are placed in the correct order, the algorithm connects the variables to achieve an appropriate dataflow.

Vivado HLS has a pragma to unroll the loops, but it has limitations. For example, if a loop containing an inner loop is imperfect, the unroll pragma at the outer loop directs Vivado HLS to replicate the inner loop by the unroll factor. This lowers the ILP in the implementation by having separate loops and limits optimization between them. Thus, unrolling them manually ensures a better hardware implementation. The user can determine the number of iterations to unfold directly or by detailing a number of load/stores. In the second case the tool analyzes the graphs and determines the load/stores that can be in parallel, and unfolds the loop enough times to have the number of simultaneous load/stores desired.

4.6 Dataflow optimizations

After Stage 4, the tool obtains a DFG that can be further optimized. In this stage, the tool currently has two types optimizations. One type focuses on arithmetic optimizations. One of these is the accumulation optimization Algorithm 9. This optimization is dedicated to writing an accumulation from partial sums. It first detects an accumulation chain and afterwards proceeds to balance it. If the chain is the length of a power of two balancing is trivial. If the length of chain is not a power of two, then at some point during the balancing process, the tool will need to connect an uneven amount of nodes. To solve this, it makes the number even by simply removing one of the nodes from the list of nodes to connect. The next time the algorithm detects an uneven amount of nodes instead of removing, the tool adds the previously removed extra node to the list thereby making the list even again. There is eventually an uneven amount of nodes since the tree ends in a single node. Figure 4.9 shows a balanced accumulation compared to the chain in Figure 4.8.

A balanced accumulation provides more ILP and can improve the execution of an algorithm. Although Vivado HLS is capable of balancing expressions it maintains the dependencies even if not necessary. For a pipeline, Vivado HLS may increase the II. As seen in Figure 4.8c the first addition depends on the result of the last sum. Therefore, the next stage of the pipeline can only initiate after the clock cycles necessary to execute that chain. However, Figure 4.9 shows that the value calculated in the previous iteration is used only once at the final sum. Therefore, the pipeline only needs to be delayed for the duration of a single sum instead of an entire accumulation chain.

Another arithmetic optimization is applied to divisions. This operation is very costly in hardware both in resources and latency so if the input code includes multiple divisions with a common divisor, then the inverse of the divisor can be calculated, and then used to implement the divisions as multiplications. For divisions with divisors which are a power of two constants the tool can simply change the operation to a right shift a constant. If that is not applicable, it identifies the source

Algorithm 7 Unfold Graph

```

1: procedure UNFOLD GRAPH(Graph G , int fold ,String type,String name)
2:   connect  $\leftarrow$  new_List()
3:   for every Node  $\in$  Graph do
4:     if  $\neg$ n.type=nop AND  $\neg$ n.type=hyper then
5:       for i $\leftarrow$ 0,fold do
6:         new_node  $\leftarrow$  n ▷ create copy of node n
7:         G.add_node(new_node)
8:         G.connect_node(new_node)
9:         for every Edge e  $\in$  new_node do
10:          if e.type=var then
11:            if e.isArray then
12:              e.update_index(name)
13:            else
14:              e.append_label(i)
15:            end if
16:            if e.hasConnect then ▷ added by stage 4
17:              connect.add(e) ▷ add updated edge
18:            end if
19:          end if
20:        end for
21:      end for
22:    end if
23:    if n.type=hyper then
24:      UNFOLD GRAPH(hyper.subgraph,fold,type, name)
25:    end if
26:  end for
27:  if type = Sequential then
28:    connect_edges(connect) ▷ Connects necessary edges between unfolded iterations
29:  end if
30: end procedure

```

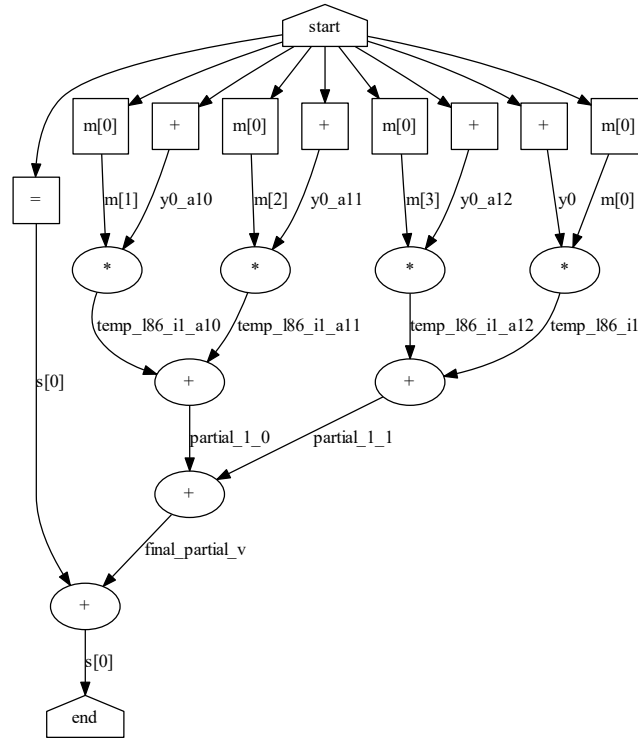
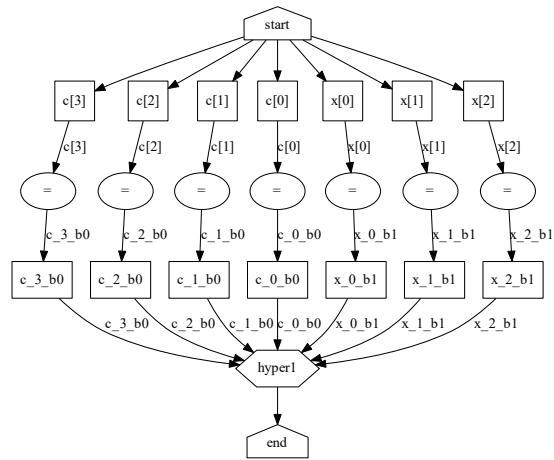


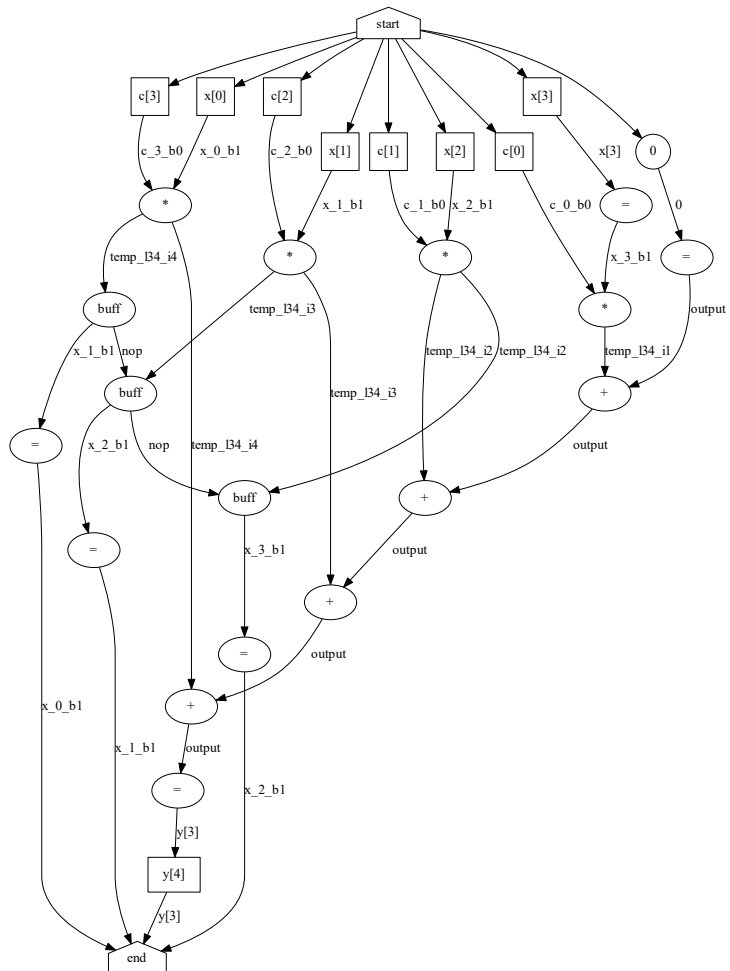
Figure 4.9: Partial sum accumulation applied to unfolded dataflow in Figure 4.8c

of the divisor. If that source is unique to multiple divisions the tool adds the nodes to calculate the inverse, and substitutes the divisions with multiplications with the inverse.

Another optimization is the parallel access optimization. In case this optimization is applied to Stage 3 loop, the Stage 4 pipelining does not occur. Memory reads in an FPGA can lead to large bottlenecks in the execution as block RAMs are limited to two simultaneous reads per cycle. Therefore, lowering the amount of reads lowers the amount of cycles needed to start executing the operations. Also memory accesses affect pipeline implementations as the next stage of the pipeline can only start after the memories are free to be used by the next stage. Thus, lowering memory accesses enable the hardware implementation to execute more efficiently. The concept behind this optimization is simple. If the next iteration of a loop accesses memory values that were used in the current iteration then those values should be stored in buffers between iterations. Algorithm 8 demonstrates the approach. First, the tool identifies the reusable accesses by comparing the accesses of one iteration with the accesses of the next. If it identifies accesses it can reuse, the tool creates the buffers for storing the values. The tool inserts the necessary nodes to read the values of the first iteration before the loop. The tool then restructures the loop to use the buffers. The tool also structures the loop to load the buffers with the values for the next iteration in the correct order. The tool adds appropriate edges and "buff" nodes to ensure that the correct dependencies are maintained.



(a) Out of loop dataflow



(b) Compact dataflow of data optimized loop

Figure 4.10: Result of memory accesses optimization for the benchmark *1D fir*

Figure 4.10 shows the end result for the benchmark *1D fir*. The original code for *1D fir* is described in Listing A.4. This optimization was applied to a filter with 4 coefficients. Thus, any output requires 4 input values to be calculated. As 3 of those values were already used in the previous iteration they can be reused. Listing 4.5 shows the resulting code from the optimizations. In the optimized code, the buffers are loaded with the values for the first iteration. Inside the loop, the new values are read and at the end of the loop the buffers are filled with the values for the next iteration. This optimization changes the structure of the Stage 3 loop. Beforehand the iterations were independent, but now they need to be executed in order, as the buffers need to be filled with new values before advancing to the next iteration. Another crucial difference is the use of "buff" type nodes to impose correct code order. This can be seen on the left side of Figure 4.10b. Without "buff" nodes, the operations $temp_l34_i4 = x_0_b1 * c_3_b0$; and $x_0_b1 = x_1_b1$; can happen in parallel since all of the necessary values for the operation are available since the start of the loop. But those lines of code are not interchangeable as that would lead to a very different execution. Additionally, the buffers need to be filled in the correct order. All the operations $x_0_b1 = x_1_b1$; $x_1_b1 = x_2_b1$; $x_2_b1 = x_3_b1$; can execute in parallel and therefore without "buff" nodes the output code can have these statements in any order. However, only a specific order leads to the correct implementation. So, the tool adds the buff nodes and connects them, so that the new value can only be loaded after the old one has been used, and the correct order for loading the buffers is maintained.

```
void fir_b_data_opt(int x[256],
int y[256], int c[4]){
    ...
    \\calculation of y values
    ...
    c_0_b0=c[0];
    ... \\preload of buffers
    x_0_b1=m[0];
    x_1_b1=m[1];
    x_2_b1=m[2];
    ...
    for( int i =0; i < 252; i=i+1){
    #pragma HLS pipeline
        x_3_b1=x[i+3];
        temp_l34_i1=x_3_b1 * c_0_b0;
        temp_l34_i2=x_2_b1 * c_1_b0;
        temp_l34_i3=x_1_b1 * c_2_b0;
        temp_l34_i4=x_0_b1 * c_3_b0;
        output=0;
        output=output+temp_l34_i1;
        ...
        y[i+3]=output;
        x_0_b1=x_1_b1;
```



```
x_1_b1=x_2_b1
x_2_b1=x_3_b1;
}
}
```

Listing 4.5: *ID fir* output code after meory access optimizations in Stage 5

Another optimizations the user can choose is the full partitioning of arrays to remove memory bottlenecks. The previous optimization by the tool reduced memory accesses through data reuse. However, the DFG might still contain a lot of memory accesses that slow the execution of the hardware implementation. Another way of lowering the memory bottleneck is through array partitioning directives. Previously, the tool set this directive based on the minimum amount of load/stores the user demanded. Based on that amount the tool unrolls the loops in the DFG, so that every single array meets that minimum. However, when doing so some arrays may have more accesses than the defined minimum, and therefore not all the accesses in the dataflow can be scheduled in one clock cycle by the HLS tool. If the user chooses to fully partition the arrays the tool makes a final pass through the whole DFG after it was unrolled and optimized. At every hierarchy level, the tool counts the number of accesses made to the arrays. It compares the amounts at every level and based on the highest number, the tool sets the appropriate array partitioning factor. These optimizations can significantly increase the resource usage. First because it increase the amount of concurrent memory reads by reproducing the original array in smaller separate BRAMs. Many FPGA BRAMs have a number of ports to be accessed so by separating an array into multiple BRAMs more concurrent accesses are possible. Another reason for the resource increase is the added logic to organize and connect the multiple BRAMs. Also by lowering the memory bottleneck more operations can be implemented in parallel, thereby increasing the amount of LUTs and DSPs required. So, although this optimizations is capable of enabling large speedups, it is not appropriate if resource usage is limited.

4.7 C code generation

The last stage generates the C code with Vivado HLS directives. First the tool passes through the graph to identify all the added variables so they can be initiated correctly. Through this pass the tool also adds a wire number to every edge that is not a input and output. After that, the tool levels the graph. This leveled graph is used to write the output. The tool has to use the leveled graph to ensure that operations are written in correct ordered. Operation of the same level are parallel and the order does not matter. However, the tool must write out the levels in order to ensure that the C output uses the correct values at the correct operations. Algorithm 10 illustrates the approach to writing. At the start, the tool initializes the function header and all the local variables if it is at the upper level. If the tool is on a lower level it instead initiates a loop. In theses initial steps the tool also injects the HLS directives. At the upper level the memory partition directives are inserted and in the loops the pipeline directive is placed. As for the operation itself the tool passes through the leveled graph. If a node is an operation node the tool writes out the corresponding statement. If

Algorithm 8 Access Optimization

```

1: procedure ACCESS OPTIMIZATION(Graph G, graphlist upperlevel )    ▷ G has to be a loop
2:   for every node  $n \in G$  do
3:     for every entering edge  $e \in n$  do
4:       if  $e.isArray$  then
5:          $curr\_access.add(e)$ 
6:          $e' \leftarrow e.increment\_index$     ▷ Get the index of the access at next iteration
7:          $next\_access.add(e')$ 
8:       end if
9:     end for
10:     $order\_access(curr\_access, next\_access)$ 
11:     $check\_redundant\_access(curr\_access, next\_access)$ 
12:    for  $i \leftarrow 0, curr\_access.size$  do
13:      if  $curr\_access(i).redundant$  then
14:        store value in buffer before the loop
15:      end if
16:      change label of  $curr\_access$  to buffer    ▷ update label use buffer value
17:      if  $\neg curr\_access(i).redundant$  then
18:        read non redundant value too buffer
19:        connect current buffer to next    ▷ update buffers by shifting
20:      end if
21:    end for
22:  end for
23: end procedure

```

Algorithm 9 Partial Sum Accumulation

```

1: procedure PARTIAL SUM ACCUMULATION(Graph G , String var)
2:   for every Node n  $\in$  G do
3:     if pred(n).name = succ(n).name then
4:       DETECT_CHAIN_UP(pred(n),G)
5:       DETECT_CHAIN_DOWN(succ(n),G)
6:       G.remove_node(n)
7:     end if
8:   end for
9:   repeat
10:    if nodes_to_connect is odd then                                ▷ Only an even amount can be summed
11:      if extra = false then
12:        extra_node  $\leftarrow$  nodes_to_connect.tail()                ▷ make even by removing a node
13:        nodes_to_connect.remove_tail()
14:        extra  $\leftarrow$  true
15:      else
16:        nodes_to_connect.add(extra_node)                          ▷ make even by adding a node
17:        extra_node  $\leftarrow$  null
18:        extra  $\leftarrow$  false
19:      end if
20:    end if
21:    for k  $\leftarrow$  0 : k+2 : nodes_to_connect/2 do
22:      G.addNode(sum)
23:      G.addEdge(nodes_to_connect(k))
24:      G.addEdge(nodes_to_connect(k+1))
25:      next_iteration.add(sum)
26:    end for
27:    nodes_to_connect.clear()
28:    nodes_to_connect=next_iteration
29:    next_iteration.clear()
30:  until nodes_to_connect only has one node AND extra = false
31:  connect last sum to end of chain
32: end procedure

33: procedure DETECT_CHAIN_UP(Node n, Graph G)
34:   if n has input edge e with same name as output edge then
35:     nodes_to_connect(input_node(e))                                ▷ add other input to list
36:     Dectect_chain_up(pred(n))
37:     G.remove_node(n)
38:   end if
39: end procedure

40: procedure DETECT_CHAIN_DOWN(Node n,Graph G)
41:   if n has input edge e with same name as output edge then
42:     nodes_to_connect(input_node(e))                                ▷ add other input to list
43:     Dectect_chain_up(succ(n))
44:     G.remove_node(n)
45:   end if
46: end procedure

```

one of the edges of the operation is a vector the tool needs to change the index. The tool changes the index to depend on the loop iteration variables if necessary. The tool does so by checking the attributes that the tool added during the matching algorithms. If the node is not an operation node but a "hyper" node, the tool extracts the DFG the hyper node contains, and passes it through Algorithm 10. Once the output is generated, the backend terminates.

4.8 Limitations

The backend still has limitations. Currently, it cannot handle conditional statements. Conditional statements are important and at the moment they cannot be implemented by our tool. Another limitation is C structures, as the backend does not correctly parse the ways they are called.

The tool also only optimizes kernels and generates loops but has no way of generating multiple separate functions and connecting them, which is important to build optimized dataflows for certain applications. Another limitation is that dataflows are good representations for hardware implementations, but they lack a way maintaining execution order which is important for situations such as the data optimizations in Stage 5. That case was solved through the usage of "buff" nodes, but those additions still force the dataflow to awkward graphs that limit other optimizations. Although DFGs are good representations of hardware, the rules of C code implementations might not align with such a representation.

There are also limitations in the way the graph applies the matching algorithms in Stage 4 and 3. Execution traces can be really big and the tool matches sequences based on certain assumptions. Considering that the input algorithm has 2 loops writing on the same array. For example, if one of the loops writes on even numbers and the other on odd, the tool would be incapable of compacting the resulting trace. So, the folding technique lacks some flexibility.

4.9 Summary

This chapter presented the implementation of the code restructuring at the backend in-depth. The backend is responsible for restructuring the graph output from the frontend to generate optimized C code. The next sections detailed in depth the steps taken at every stage, and presented the resulting DFGs and code from applying the algorithms of the different stages to some of the benchmarks. These sections show how the backends folds the initial unfolded DFG and then how the tool optimizes the compacted dataflow, to generate C code that shows the desired properties for an HLS tool. The backend automatically restructures the input graph leading to a C output that might be very different from the original C input code, based on a few configurations from the user. The final section presented some limitations of the tool.

The current version of the backend is capable of handling many different types of input DFGs and automatically generating an optimized C code with directives. This result is achieved by passing through a sequence of stages that first compact the graphs and subsequently restructure it to obtain an improved representation. The current improvements in the tool are achieved through

Algorithm 10 Generate C code

```

1: procedure GENERATE_C_CODE(leveledgraph G1 , boolean upper)
2:   if upper=true then
3:     write function header
4:   else
5:     write loop header
6:   end if
7:   for every level l  $\in$  G1 do
8:     for every node  $\in$  l do
9:       if n.type = op then
10:        WRITE_OP(n,upper)
11:       else if n.type = hyper then
12:        GENERATE_C_CODE(n.subgraph.leveledgraph, not upper)
13:       end if
14:     end for
15:   end for
16: end procedure

17: procedure WRITE_OP(Node n,upper)
18:   result  $\leftarrow$  n.outgoing_edge.label
19:   op  $\leftarrow$  n.label
20:   if op.label = " = " then ▷ In case of attribution
21:     input  $\leftarrow$  n.entering_edge.label
22:     if  $\neg$ upper then
23:       index_for_loop(result)
24:       index_for_loop(input)
25:     end if
26:     write_out(result = input,outputfile) ▷ Write C in outputfile
27:   else
28:     input_right  $\leftarrow$  n.entering_edge_right.label
29:     input_left  $\leftarrow$  n.entering_edge_left.label
30:     if  $\neg$ upper then
31:       index_for_loop(result)
32:       index_for_loop(input_right)
33:       index_for_loop(input_left)
34:     end if
35:     write_out(result = input_left op input_right,outputfile)
36:   end if
37: end procedure

```

identifying potential pipelines in the DFG, optimizing memory accesses and optimizing arithmetic importation in the DFG. These optimizations can greatly restructure the DFGs as shown by the multiple iterations of the DFGs that this chapter shows. These DFGs correspond to output source codes that implement the same algorithm as the original code but are structured very differently depending on the chosen input configurations. Although, the backend is more developed than the frontend, the backend still has limitations. Like the frontend, it cannot handle conditional statements, and it does not handle function calls inside the DFGs. Despite these limitations, the backend is still versatile and can output many different kinds of optimized C code automatically with some simple configurations.

Chapter 5

Experimental Results

This chapter outlines the results obtained by the tool. The approach is applied to a series of benchmarks. The frontend stage was implemented manually and the backend processed the graphs and generated the C code with directives automatically. The results are dependent on the input configurations.

5.1 Experimental setup

All the benchmarks consist of operation heavy algorithms with very little control flow. Tables 5.1 and 5.2 detail information about the benchmarks. They represent DSP algorithms. The benchmarks are either from the DSPLIB from Texas Instruments [32], the UTDSP Benchmark Suite [33] or from an MPEG audio encoder [31]. The *SVM* kernel is the one presented in [24]. The simplest benchmark is the *dotproduct* from DSPLIB. The *Autocorrelation* benchmark from DSPLIB is also used. The *1D fir* benchmark is a typical code implementing a FIR filter with N taps. The *filter subband* benchmark comes from a part of an MPEG audio encoder. *2D Convolution* is the largest benchmark which is a kernel that performs a 2D convolution. This convolution is part of the Sobel edge detection application from UTDSP. The dataset sizes are identical to the original benchmarks except for the *dotproduct*, in which larger input vector is used, and for the *2D Convolution*, in which the image size is decreased. The code for all benchmarks is in Appendix A.

The effectiveness of the approach is analyzed for the multiple optimization levels presented in Table 5.3. Level 01 applies no directives or code restructuring. Level 02 passes through all the stages, but does not implement Stage 5 optimizations (see Fig 3.4). Level 03 adds automated memory partitioning directives to the previous level. Level 04 applies the memory optimizations in Stage 5 (see Fig 3.4). Level 05 adds the arithmetic optimizations to Level 03, and Level 06 adds the arithmetic optimizations to level 04. Level 07 and 08 apply full array partitioning to Level 05 and 06 respectively. The results of the C code generated considering these optimizations are compared over the input C code with manual optimizations. These C code versions are briefly summarized in Table 5.4. Without directives, the implementation in FPGAs is very unoptimized, resulting in achievements by the tool of better results with very little work. It is a fair assumption

| Benchmark | Benchmark Source | Dataset Size | Brief Description |
|------------------------|------------------|--------------------------------------|---|
| <i>filter subband</i> | MPEG [31] | Ns =32 Ny=64 Nz=512 Nm 1024 | Compresses an Nz sized input into Ny sized vector. Filters this vector with Ny coefficients of an Nm sized vector to an Ns sized output |
| <i>Autocorrelation</i> | DSPLIB [32] | sd =170 ac=10 | Calculates the autocorrelation value of a given vector |
| <i>dotproduct</i> | DSPLIB [32] | N=2000 | Calculates the dot product of two equal sized vectors |
| <i>1D fir</i> | in-house | N=257 Nc=32 | A finite-impulse response (FIR) filter considering N input samples and Nc taps |
| <i>2D Convolution</i> | UTDSP [33] | K=3 N=64 | Convolve NxN input image using a KxK kernel |
| <i>SVM</i> | Paper [24] | N_sv=1248 D_sv=18 | Support Vector Machine (SVM) kernel applied to arrhythmia detection |

Table 5.1: Benchmark information

| Benchmark | Nr Lines of code Nr. of loops | Nested Structure | Nr. of Node | Nr. of Edges |
|------------------------|----------------------------------|--|--------------------|--------------------|
| <i>filter subband</i> | 16 4 | Two sets of nested loops | 10,976 6402 | 13,920 11,040 |
| <i>Autocorrelation</i> | 10 2 | A single nested loop set | 6779 3581 | 9640 6799 |
| <i>dotproduct</i> | 9 1 | A single loop | 12,003 8006 | 12,002 12,005 |
| <i>1D fir</i> | 12 2 | A single nested loop set | 2051 1542 | 2050 2053 |
| <i>2D Convolution</i> | 45 4 | A single nested loop set with 4 levels | 157,892 88,693 | 222,986 165,584 |
| <i>SVM</i> | 145 2 | A single nested loop set | 168,503 192,195 | 97,369 146,041 |

Table 5.2: Main Frontend DFG information for each benchmark

| Optimization level of the framework | Brief Description |
|-------------------------------------|--|
| 01 | No optimizations on the graph |
| 02 | Input graph is folded as much as possible, and unfolded according to user configurations |
| 03 | Adds array partitioning to level 02 |
| 04 | Adds data reuse to level 03 |
| 05 | Adds arithmetic optimizations to level 03 |
| 06 | Adds arithmetic optimizations to level 04 |
| 07 | Adds full array partition to level 05 |
| 08 | Adds full array partition to level 06 |

Table 5.3: Description of framework levels

that a software programmer could use some very basic directives. However, one cannot assume a typical software programmer is proficient with all types of directives. Thus, this approach to the evaluation allows us to perceive the effectiveness of the tool for different levels of hardware design knowledge. In certain cases more directives lead to worse implementations. In those cases C-high and C-inter coincide.

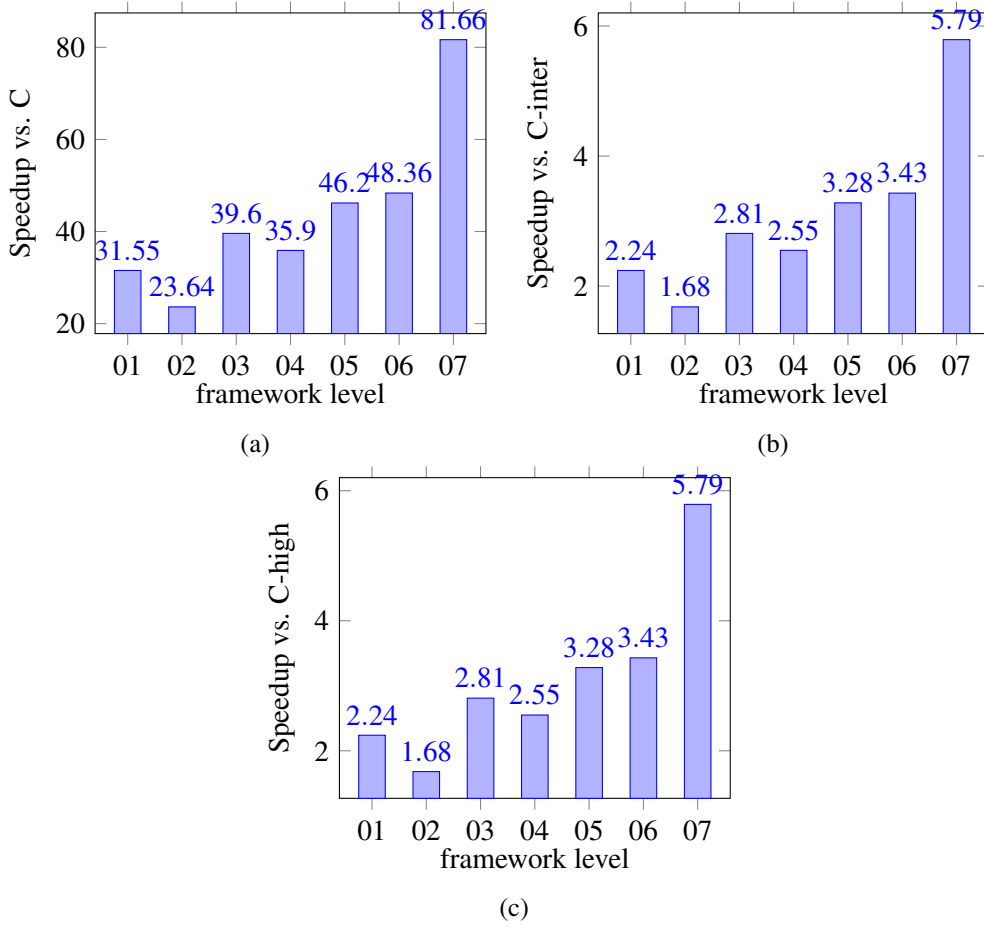
The speed and resource values are the ones presented in the reports resulting from synthesizing the C code with Vivado HLS 2017.4, and targeting a Artix™-7 FPGA Xilinx (xc7z020clg484-1). The tool was executed in a PC with an Intel Core i7-7700 with 32GB RAM. From the reports we take for every benchmark the number of LUTs, DSPs, BRAMs and FFs, as well the Latency, which is the number of clock cycles necessary to complete the kernel in hardware, and the duration of the clock cycles, which are in nanoseconds. All of the benchmarks had a time constraint of 10ns except *filter subband* which has a constraint of 20ns. The total execution time of a hardware implementation is calculated as the multiplication between the max frequency and the latency indicated in the post synthesis report. The speedups are the result of dividing the total execution time of the implementations from Table 5.4 by the total execution time of the results of different framework optimizations levels. The correctness of the C code generated by the tool is evaluated using Vivado HLS. Vivado HLS allows both the C as well as the resulting RTL description to be validated. Both the C code as well as the RTL implementations are validated through testbenches. All of the implementations passed the Vivado HLS evaluation.

5.2 Results

The *filter subband* benchmark is a very good example for showing the best aspects of the tool since at every level the tool presents speedup gains even compared to the C-high. Figure 5.1 shows the speedups relative to the original C, C-inter and C-high. Level 01 has a large speedup because it exposes the most ILP since it has no loops and is completely unrolled. However its resource usage greatly surpasses the maximum resources of the FPGA used. To limit the resource usage

| Comparison code | Brief Description |
|-----------------|---|
| C | Original code without any modifications |
| C-inter | Input code optimized with basic directives such as the pipeline directive |
| C-high | Improved C-inter implementation with unroll and memory partition directives |

Table 5.4: Description of the different versions of input code used to compare results

Figure 5.1: Speedups for *filter subband*

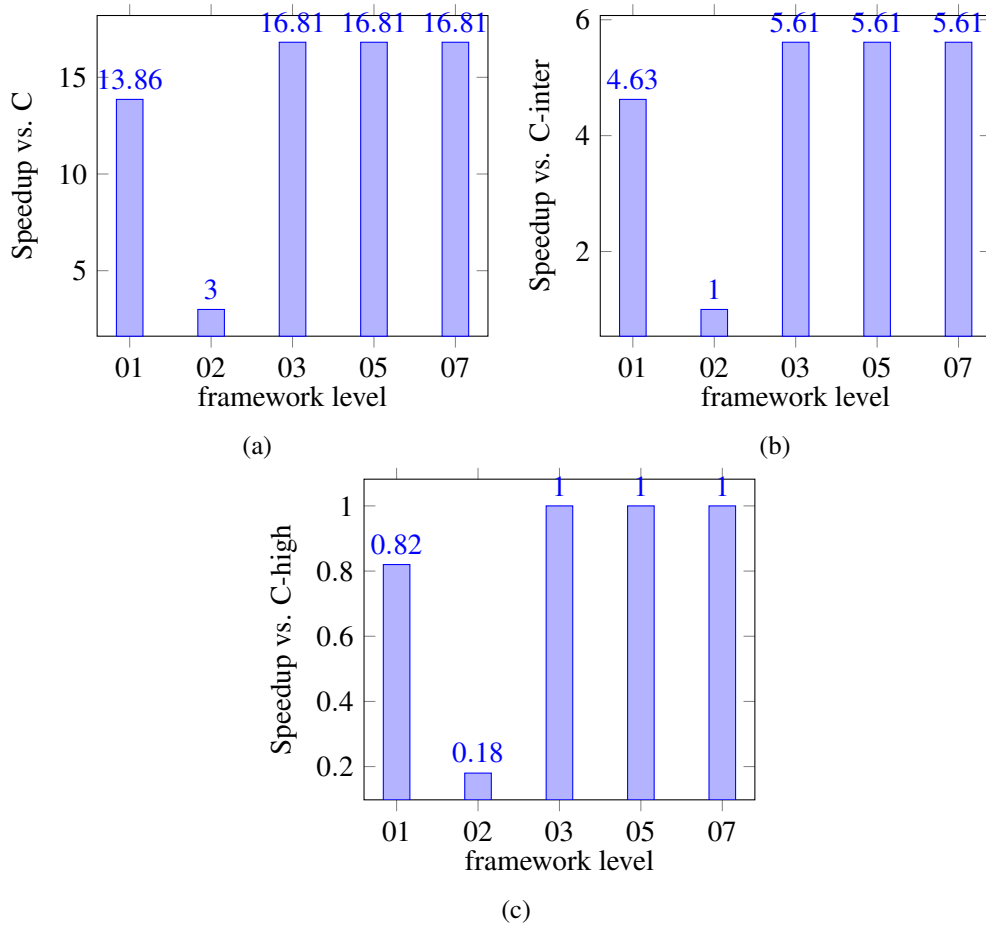
the tool compacts the dataflow in loops. By folding in level 02 the tool achieves improved results compared to C-high. The configuration for the folding for this benchmark are four load/stores and high folding. This speedup is due to the pipeline the tool generates as seen in Listing 4.4 which performs the algorithm more efficiently. In level 03 the tool the speedup increases to 2.81 \times times compared to the C-high version. Level 04 optimizations result in a speedup of 2.55 times compared to C-high. This speedup is due to the fact that every output calculation reuses half of the data that was used for the previous iteration. Thus, by reusing data the tool halves the memory reads per iteration. This has a large impact on the pipeline since the tool can lower the initiation value (II) of loop pipelining by lowering the amount of memory reads. The optimized loop is

| Source | LUT | FF | DSP | BRAM | Latency | Time(ns) |
|--------------|-------------------|-------------------|-------|------|---------|----------|
| C | 2,050 | 1,115 | 14 | 2 | 23,746 | 17.22 |
| C-inter | 6,633 | 11,911 | 28 | 98 | 1,581 | 18.34 |
| C-high | 6,633 | 11,911 | 28 | 98 | 1,581 | 18.34 |
| framework-01 | 38,132 | 77,945 | 162 | 0 | 648 | 20 |
| framework-02 | 9,480 | 19,196 | 31 | 0 | 1,083 | 15.97 |
| framework-03 | 12,065 | 18,840 | 59 | 0 | 563 | 18.34 |
| framework-04 | 23,746 | 45,107 | 112 | 321 | 621 | 18.34 |
| framework-05 | 12,676 | 23,554 | 59 | 0 | 557 | 15.89 |
| framework-06 | 23,291 | 30,074 | 112 | 0 | 461 | 18.34 |
| framework-07 | 47,537 | 42,598 | 118 | 0 | 293 | 17.09 |
| framework-08 | $4.41 \cdot 10^5$ | $2.22 \cdot 10^5$ | 3,584 | 0 | 74 | 17.22 |

Table 5.5: HLS results for *filter subband*

the stage 3 loop so the stage 4 pipeline cannot be implemented. Level 05 obtains a speedup of 3.28 times compared to C-high. This is a significant increase compared to $2.81\times$ in level 03. The arithmetic optimizations do not heavily affect the latency or the II of the implementation. However, by separating the accumulation chains Vivado HLS synthesizes the code differently. Previously to execute the chained sums Vivado HLS applied adders that are chained together to be more efficient. However, with partial sums Vivado HLS synthesizes addition in parallel. The implementation of these adders is different and therefore the result has a lower frequency that leads to the larger speedup. By adding arithmetic optimizations in level 06 a speedup of $3.43\times$ is obtained. Due to the fact that previously all results of the sums were saved in the output vector Vivado HLS had to write in memory every middle values which is unnecessary. By balancing this chain and storing the result in local variables delays caused by memory accesses are removed. Level 07 obtains a speedup of $5.49\times$ compared to C-high, because by partitioning the memory more ILP is possible. Level 08 has a very large speedup but is not included since the necessary resource are far beyond the capabilities of the target FPGA. When it comes to resource usage all the implementations use more resources since they have more ILP (see Table 5.5). However, they use fewer BRAMs because the output code no longer uses a local array, and instead stores the values in registers. In the previous method, Vivado HLS dealt with two different loops so it had to store the values, calculated in the first loop, in memory before start the new loop. The only exception is the level 04 that uses a lot of BRAMs. This is because it stores the results of every sum in the output vector. So to allow pipelining it requires to instantiate additional BRAMs to store the result of the sums to ensure values are not lost. However, by partitioning the sums in level 06, Vivado HLS only stores in the output vector at the end. So it does not need any additional BRAMs and instead saves them in registers.

The *dotproduct* is a simple algorithm. For the benchmark 8 load/stores and high folding is used. The first levels do not lead to better results. But once the tool applies memory partitioning it matches the speed of the C-high version as seen in Figure 5.2 and Table 5.6. The output of level 03 is faster then the basic and inter versions of the input with a $16.8\times$ and a $5.6\times$ speedup,

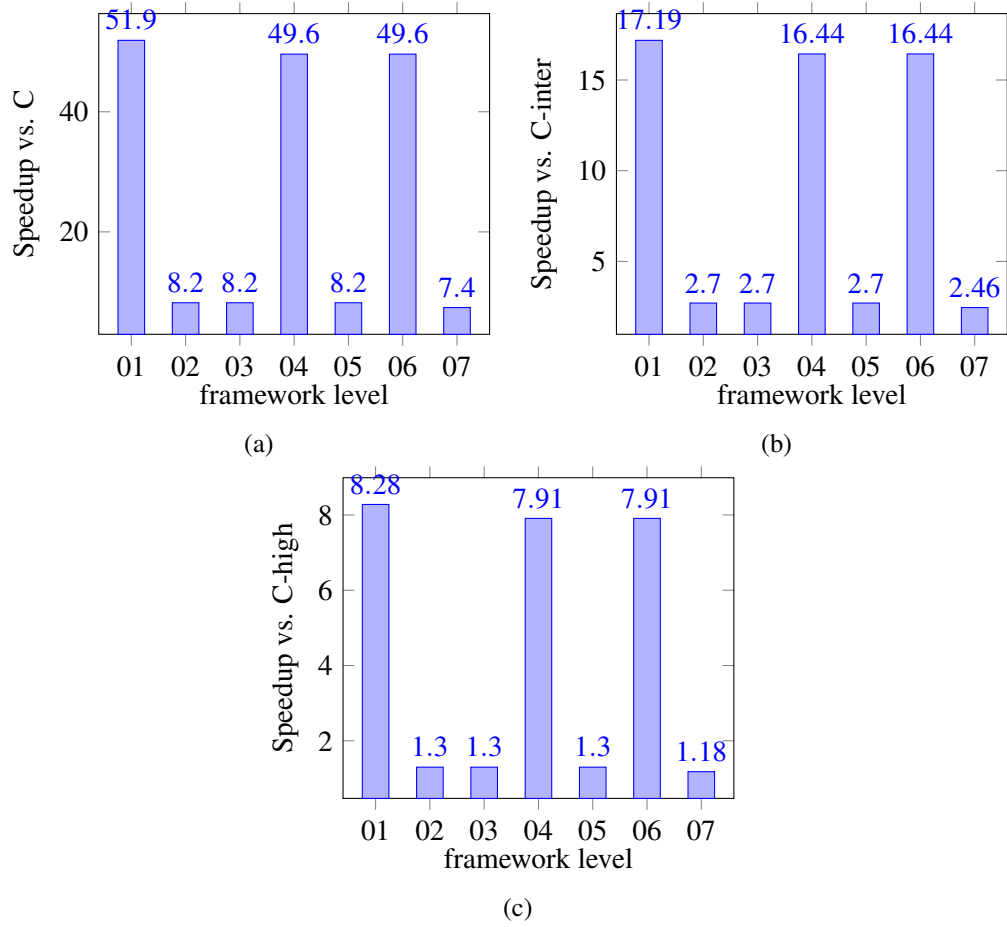
Figure 5.2: Speedups for *dotproduct*

| Source | LUT | FF | DSP | BRAM | Latency | Time(ns) |
|--------------|--------|--------|-------|------|---------|----------|
| C | 76 | 90 | 1 | 0 | 6,001 | 6.38 |
| C-pipe | 104 | 83 | 1 | 0 | 1,003 | 8.7 |
| C-high | 294 | 435 | 8 | 0 | 254 | 8.93 |
| framework-01 | 51,284 | 62,338 | 2,000 | 0 | 257 | 10.75 |
| framework-02 | 104 | 83 | 1 | 0 | 2,003 | 6.38 |
| framework-03 | 294 | 581 | 8 | 0 | 255 | 8.93 |
| framework-05 | 294 | 581 | 8 | 0 | 255 | 8.93 |
| framework-07 | 294 | 581 | 8 | 0 | 255 | 8.93 |

Table 5.6: HLS results for *dotproduct*

respectively. Without memory redundancy level 04 does not change results. All the higher levels do not add anything impact-full, therefore the speedups do not change for levels 03, 05 and 07.

Autocorrelation is another kernel that shows very interesting results (see Figure 5.3). It consist of a small outermost loop with a big internal loop. The configuration for the folding in this benchmark are four load/stores and high folding. The tool already obtains positive results by level 02 with 1.3× gain compared to C-high and 2.7× compared to C-inter. This is because of the inner-

Figure 5.3: Speedups for *Autocorrelation*

| Source | LUT | FF | DSP | BRAM | Latency | Time(ns) |
|--------------|--------|--------|-------|------|---------|----------|
| C | 106 | 107 | 1 | 0 | 6,421 | 6.38 |
| C-inter | 400 | 310 | 1 | 0 | 1,604 | 8.46 |
| C-high | 1,986 | 894 | 10 | 0 | 1,643 | 7.66 |
| framework-01 | 41,868 | 31,170 | 1,600 | 0 | 853 | 7.66 |
| framework-02 | 930 | 459 | 10 | 0 | 655 | 7.66 |
| framework-03 | 1,117 | 603 | 10 | 0 | 655 | 7.66 |
| framework-04 | 9,083 | 7,277 | 160 | 0 | 655 | 7.66 |
| framework-05 | 1,117 | 587 | 10 | 0 | 655 | 7.66 |
| framework-06 | 9,083 | 7,277 | 160 | 0 | 96 | 8.6 |
| framework-07 | 1,282 | 3,567 | 11 | 0 | 669 | 8.26 |
| framework-08 | 8,025 | 7,114 | 160 | 0 | 16 | 8.6 |

Table 5.7: HLS results for *Autocorrelation*

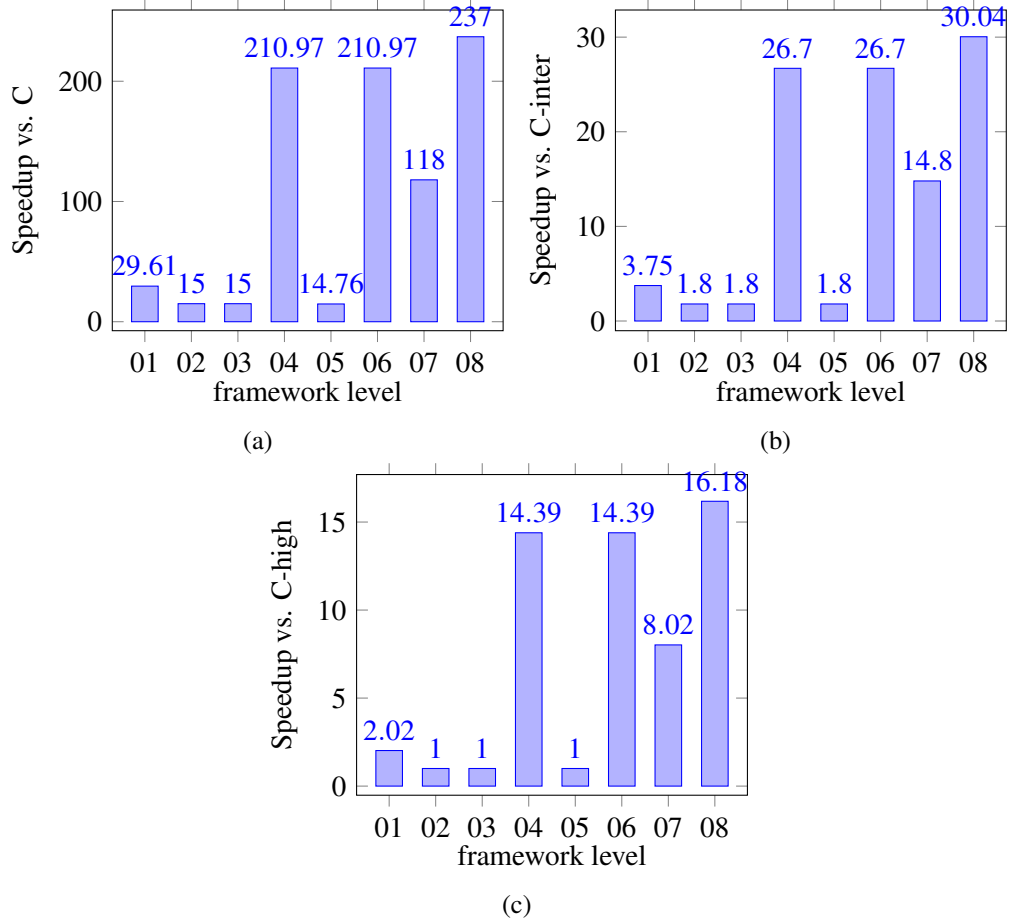
most loop, since the unroll directive on the outer loop does not consider loop fusion of the unrolled code. The directive generates multiple independent copies of the inner loop which has pipeline directives. The manual unfolding combines them in a single loop and exposes more ILP. It has

many benefits in the *Autocorrelation* application which has a lot of redundant memory use. If the pipelines are separated then Vivado HLS does not take advantage of these redundant accesses and schedules many more memory reads. This improved loop unrolling capacity showcases another superiority of the DFG approach as it allows us to generate better unrolled innermost parallel loops, since it is only needed to replicate the dataflow and continue having a single inner loop. As mentioned, this application has a lot of redundant memory accesses so it is a prime target for the level 04 optimization. When the memory usage is optimized, the implementation has an increase in performance. It is $7.9\times$ times faster than the highly optimized C input and $16.44\times$ compared to C-inter. Of course this comes at a cost of a large increase in resource usage (see Table 5.7). This increase is due to the fact that by applying this optimization, the innermost loop of the kernel is fully unrolled. The arithmetic optimizations in Level 06 do not increase the speedup relative to level 04. Level 08 is not included in Figure 5.3 because it reaches a speedup of $47.49\times$ compared to C-high which is far bigger than the rest. This is because the tool completely partitions the input array, and in the case of the Level 06 *Autocorrelation* benchmark a lot of cycles were dedicated to reading the sd values before starting the loop. Reading them all in a single cycle hugely affects the output. However, this level of partitioning is possible because the autocorrelated vector is a small input vector of 170 integer values. Larger vectors cannot be fully partitioned. Level 08 does not increase a lot the resource usage compared to level 06.

| Source | LUT | FF | DSP | BRAM | Latency | Time(ns) |
|--------------|-------------------|-------------------|--------|------|---------|----------|
| C | 206 | 170 | 3 | 0 | 29,251 | 8.51 |
| C-inter | 1,556 | 702 | 6 | 0 | 3,605 | 8.74 |
| C-high | 4,623 | 2,968 | 117 | 0 | 1,942 | 8.74 |
| framework-01 | $5.37 \cdot 10^6$ | $3.03 \cdot 10^5$ | 19,155 | 0 | 962 | 8.74 |
| framework-02 | 3,945 | 2,912 | 126 | 0 | 1,929 | 8.74 |
| framework-03 | 3,945 | 2,912 | 126 | 0 | 1,929 | 8.74 |
| framework-04 | 4,587 | 6,579 | 192 | 0 | 135 | 8.74 |
| framework-05 | 3,945 | 2,912 | 126 | 0 | 1,929 | 8.74 |
| framework-06 | 4,587 | 6,579 | 192 | 0 | 135 | 8.74 |
| framework-07 | 12,774 | 13,136 | 192 | 0 | 242 | 8.74 |
| framework-08 | 4,297 | 5,641 | 192 | 0 | 120 | 8.74 |

Table 5.8: HLS results for *1D fir*

The *1D fir* benchmark shows the impact of Stage 5 optimizations. For this benchmark medium folding and 2 concurrent load/stores were set in the configuration files. By merely folding in Level 02 and 03 near identical results to the C-high version are obtained, as seen in Figure 5.4. This is because the tool identifies the same loops as the original. The output of level 03 only has a speed gain of $1.86\times$ compared to C-inter. Once the tool optimizes the accesses in Level 04 the speedup is of $14.39\times$ compared to C-high version and $26.7\times$ compared to the C-inter. This is a significant improvement to an already optimized implementation. In this benchmark 32 coefficients are used. Thus, 32 inputs are needed to calculate a new output. However 31 are reused from the previous iteration so only one new value is read with this optimization, leading

Figure 5.4: Speedups for *1D fir*

to the significant performance increase. This optimization enables the pipelined loop with a II of 1 compared to 17, which is consistent with the near 17 \times speedup of 14.39 \times . Again the resource usage is increased by the exposure of more ILP (see Table 5.8). The most noticeable increases are in FFs to store more values and in DSPs to do more concurrent multiplications. The arithmetic optimizations of Level 06 do not impact the speedup compared to 04. Level 07 optimization have a large impact reaching a speedup of 8 times compared to 1 \times of 05. This is due to memory partitioning minimizing the memory bottleneck. Partitioning the memory in Level 08 increases the speedup of 06 to 16.18 \times compared to C-high.

As in some of the the previous cases, for the *2D Convolution* benchmark Level 02 and 03 generate identical loops so there is no speedup compared to C-high and 1.6 \times times speedup compared to C-inter (see Figure 5.5). 2 load/stores and high folding were the configuration for this benchmark. With a 3 \times 3 kernel every time an new pixel is calculated it is necessary to have the values of the 9 adjacent pixels. However 6 of those were used in the calculation of the previous pixel so only 3 new values are needed. By applying data reuse at Level 04 a speedup of 1.36 \times compared to C-high and 2.25 \times compared to C-inter is achieved. In this case the II achieved for the pipeline of the inner loop was 3 instead of 6 due to less memory accesses. The speedup is not

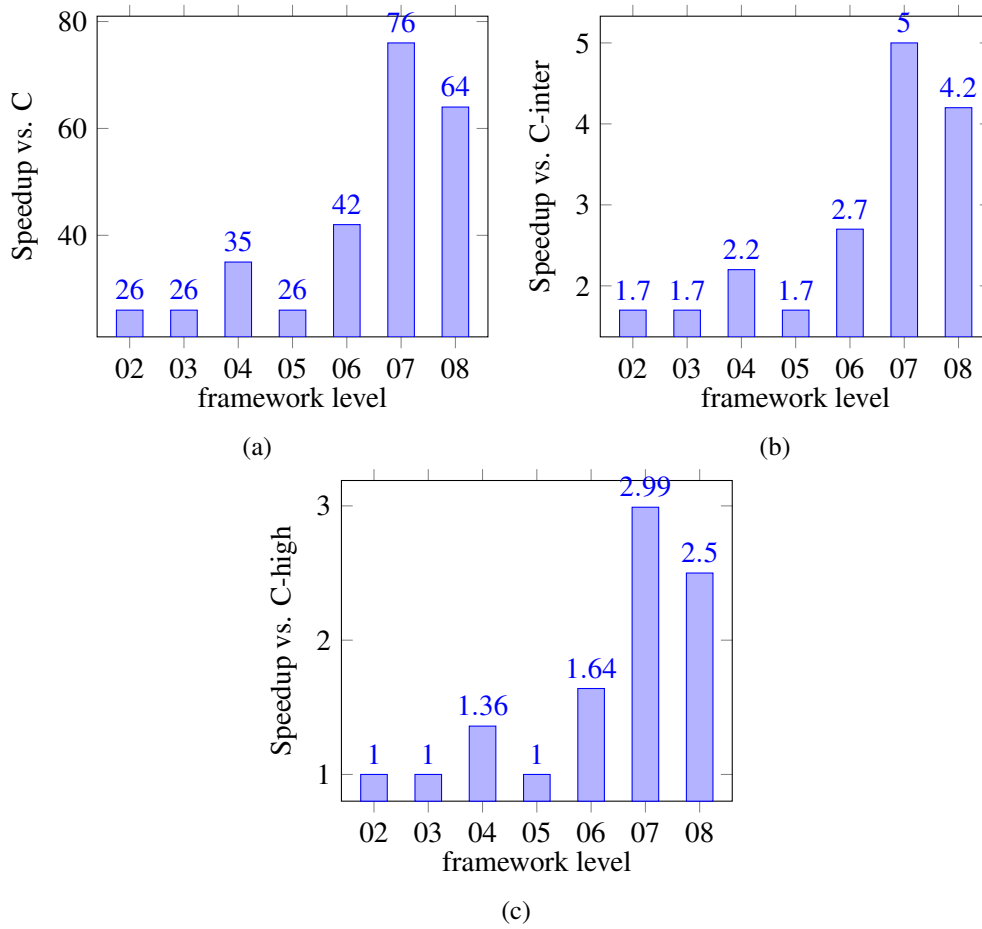


Figure 5.5: Speedups for 2D Convolution

| Source | LUT | FF | DSP | BRAM | Latency | Time(ns) |
|--------------|-------|-------|-----|------|-------------------|----------|
| C | 994 | 721 | 3 | 0 | $3.04 \cdot 10^5$ | 8.51 |
| C-inter | 277 | 2,977 | 6 | 0 | 19,294 | 8.74 |
| C-high | 5,169 | 5,833 | 36 | 0 | 11,606 | 8.74 |
| framework-02 | 5,249 | 6,000 | 39 | 0 | 11,578 | 8.74 |
| framework-03 | 5,249 | 6,000 | 39 | 0 | 11,578 | 8.74 |
| framework-04 | 5,354 | 6,575 | 54 | 0 | 8,563 | 8.74 |
| framework-05 | 4,750 | 3,082 | 51 | 0 | 11,577 | 8.74 |
| framework-06 | 4,085 | 3,461 | 57 | 0 | 7,097 | 8.74 |
| framework-07 | 6,376 | 3,408 | 57 | 0 | 3,886 | 8.74 |
| framework-08 | 5,862 | 3,794 | 60 | 0 | 3,125 | 12.97 |

Table 5.9: HLS results for 2D Convolution

as large as expected because by adding memory reuse the structure of the loops is changed. The *2D convolution* uses two nested loops to traverse the 2D array. In the original code there is no operations between the outer and inner loop so it is a perfect loop. By optimizing memory accesses the buffers are loaded before entering the inner loop, so the outer loop is no longer a perfect loop.

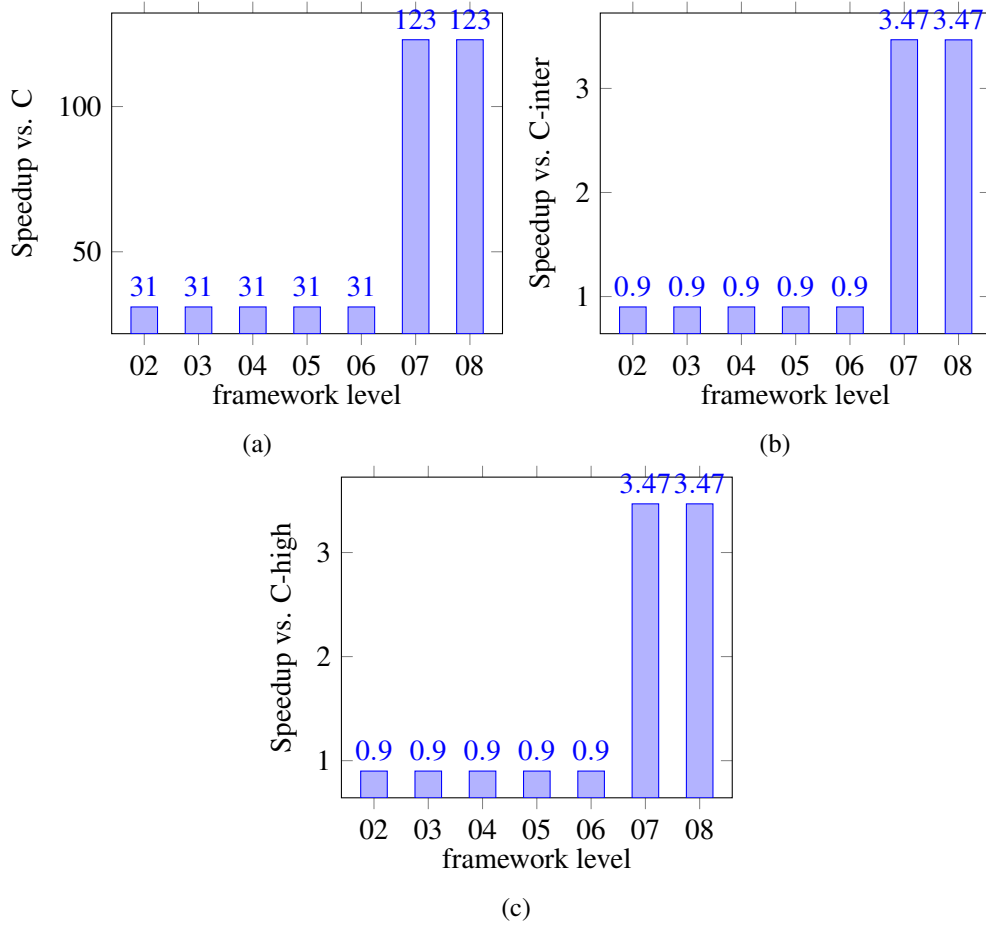
Previously Vivado HLS automatically flattened the loops and optimized the execution. Without a perfect loop that is not possibly so the improvement is not as high as expected.

In level 06 this speedup is increased by combining it with arithmetic optimizations. The differentiator here is optimizing the division in the loop. Since the divisor is common in every iteration, the tool calculates the inverse outside the loop and it substitutes the division by a multiplication with the inverse. Since multiplications are more efficient in hardware than divisions, the pipeline depth decreases by. Thus, the speedup of this level is of 1.64×. The results of Level 05 shows that arithmetic optimizations without data reuse do not have a big impact simply because of loop flattening. Like in Level 06 the iteration latency is decreased but due to loop flattening the implementation only has one loop, and lowering the iteration latency in a large pipeline with many stages does not have a large impact, because the divisions still has to be implemented outside the loop before starting the pipeline. On level 06 without loop flattening a smaller pipeline is going to be executed multiple times in a loop, so decreasing the iteration latency has a bigger impact. Another benefit of the the division optimization is that it lowers the resource usage (see Table 5.9). As partitioning the arrays in Levels 07 and 08 achieves speedups of 2.99× and 2.5× respectively. Unlike in other cases, level 07 has a better implementation than 08. This is due to the way that Vivado HLS implements the two solutions in this case. Level 08 has a pipeline with lower II and lower depth than 07. The partitioning of 08 actually performs better, but the implementation of Vivado HLS worsen the results because the inner loop is unrolled by a factor of 2 and Vivado HLS implements the last two multiplications in a single unit with larger frequency than a single multiplier. This does not happen in 07 and ,therefore the frequency is lower, leading to the higher speedup. For level 07, if the memory partition is applied but not the arithmetic ones, the speedup would only be of 2.27×, because the accumulation chains lower the effectiveness of the implementations. Therefore, it is not just a question of partitioning memory with directives. It is also necessary to restructure the code to unlock more ILP and larger speedups.

| Source | LUT | FF | DSP | BRAM | Latency | Time(ns) |
|--------------|--------|--------|-----|------|-------------------|----------|
| C | 5,923 | 3,471 | 45 | 0 | $4.04 \cdot 10^5$ | 8.23 |
| C-inter | 9,020 | 8,042 | 57 | 0 | 11,351 | 8.23 |
| C-high | 9,020 | 8,042 | 57 | 0 | 11,351 | 9.38 |
| framework-02 | 9,248 | 9,027 | 56 | 0 | 11,351 | 9.38 |
| framework-03 | 9,248 | 9,027 | 56 | 0 | 11,351 | 9.38 |
| framework-04 | 9,288 | 9,068 | 56 | 0 | 11,351 | 9.38 |
| framework-05 | 8,470 | 6,295 | 56 | 0 | 11,351 | 9.38 |
| framework-06 | 8,506 | 6,954 | 56 | 0 | 11,351 | 9.38 |
| framework-07 | 14,263 | 12,504 | 91 | 0 | 3,207 | 8.4 |
| framework-08 | 14,221 | 12,505 | 91 | 0 | 3,208 | 8.4 |

Table 5.10: HLS results for SVM kernel

Another benchmark is an SVM kernel which was also discussed in the related work section [24]. For the SVM benchmark two load/stores and high folding are used. This algorithm is an ex-

Figure 5.6: Speedups for *SVM*

ample of the implementation that is bottlenecked by memory accesses that cannot be reused. The result obtained by the tool are seen in Figure 5.6 and Table 5.10. Previously, memory redundancy was taken advantage of to allow ILP and improve the execution. Without that, it is necessary to rely on memory partitioning to be able to access more memory positions simultaneously. The *SVM* algorithm contains a very big outer loop with a smaller inner loop. It is far more preferable to fully unroll the inner loop and pipeline the outer loop since it is the bigger one and Vivado HLS cannot handle pipelines within pipelines. Also the majority of the resources come from the outer loop that calculates a floating-point exponential. But better results are obtained without unrolling. This is due to the fact that the bottleneck is all the memory reads so by unrolling ILP is not increased, and in fact the frequency is worsened. It is because of that, that levels 01 to 06 have lower performances than the original, since the tool unrolls the loop by a factor of 2 due to the fact that 2 load/reads were indicated in the configurations. These level obtain a speedup of 0.88. However, when we start maximizing the array partitioning the performance improves. In Level 07 and 08 speedup of 3.47 \times relative to C-high is achieved. If the tool did not optimize the arithmetic operation in the dataflow the speedup would only be of 1.12 \times compared to C-high. So, as with the previous benchmark, *SVM* shows that it is necessary to limit memory bottlenecks and

executional bottlenecks. Thus, continuing to show the necessity of a combination of directives and code restructuring to achieve the optimal results.

Comparing the result of the tool with that of [24], we see that the authors optimized the kernel in a very similar way. The original version of the authors code is identical to the one in Appendix A with the exception of a single final if statement that classifies the vectors based on the sum. There are many similarities in the resulting codes, depending on the optimizations. The big difference is that the tool does not partition the *SVM* kernel itself to increase concurrency. The tool attempts to obtain a similar result through unfolding the outer loop and applying array partitioning directives. The rest of the optimizations proposed in [24] are very similar. The authors balance the accumulations in a tree just like our approach. They also unroll the inner loops and apply pipeline directives like our tool. Thus, our tool automatically obtains a similar code compared to the optimized one shown in the article, depending on the users given tool configurations.

In [24] the authors present the speedups to the original C implementation for varying vector dimensions while unrolling the inner loop and balancing operations. The authors also use the same FPGAs as the one used in our evaluations. In our implementation the number of elements in the vectors are 18 and we pipelined the outer loop. In the authors implementation they partition the outer loop by factor of 2 and they unroll the inner loop. For a unroll factor of 9 and 18 elements in the vector the authors achieve a speedup of approximately 12 \times . If we extrapolate this result for a unroll factor of 18, which would be the same as our implementation, the speedup would be around 24 \times compared to our 31.2 \times in levels 02-06. Although the optimizations are similar the difference in this case is the pipelining of the outer loop, leading to our implementation being faster, as the authors do not pipeline it. However, the pipelining of the outer loops is not possible for large dimension as the inner loop would be too large and the resource intensive. The authors did not apply in that measurement array partitioning to maximize the concurrency, thus when compared with Levels 07-08 our backend tool achieves even larger speedups. The authors do not indicate resource usage.

5.3 Execution time and scalability

We measured the execution times of the backend, to further asses the results of the tool. Although the the tool improves the resulting implementations, if it takes too long to process, then it might not be a viable tool. Table 5.11 presents the execution times of the backend. For most benchmarks the time was between 1 and 2 seconds, with the exceptions of *2D Convolution* that averages between 11 and 12 seconds of execution time and *SVM* which averages between 4.5 and 6 seconds of execution time. Another exception is the level 04 for the *Autocorrelation* which executed in 5 sec. Due to the memory optimizations there is no Stage 4 folding, leading to a very large DFG in comparison to the fully folded implementations. The fastest levels are 02 and 03. Although, 01 implements no optimizations the processing of a large DFG leads to long execution times. The increase of the execution time for 04 depends on the complexity and size of the optimized loop. Level 05 to 08 do not impact the execution times compared to the previous levels, with the

exception of Level 06 and 08 for the *SVM* benchmark. These optimizations do not lead to large execution time differences because they do not significantly change the size of the DFG and they are applied to the compacted DFGs.

| Benchmark | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 |
|------------------------|-------|-------|-------|-------|-------|-------|-------|-------|
| <i>filter subband</i> | 2634 | 1944 | 1944 | 2829 | 2030 | 2791 | 2050 | 2900 |
| <i>Autocorrelation</i> | 1613 | 995 | 995 | 4757 | 1002 | 5300 | 1003 | 4985 |
| <i>dotproduct</i> | 2455 | 804 | 804 | – | 880 | – | 890 | – |
| <i>1D fir</i> | 10917 | 1695 | 1695 | 1715 | 1790 | 1793 | 1812 | 1824 |
| <i>2D Convolution</i> | – | 11408 | 11544 | 11642 | 11608 | 11783 | 11628 | 11790 |
| <i>SVM</i> | – | 4535 | 4535 | 4702 | 4700 | 5839 | 4541 | 5922 |

Table 5.11: Execution time of backend in ms for each optimization level

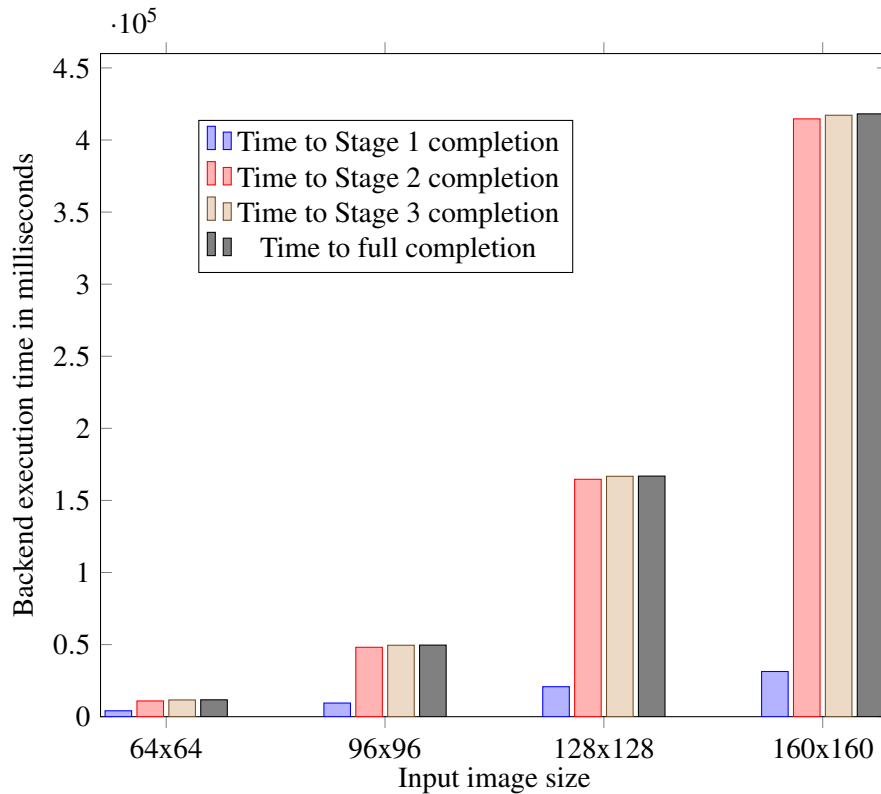


Figure 5.7: Backend execution time in seconds for multiple input image sizes

In order to analyze the impact of the dataset sizes, and consequently of the input DFG size, on the execution time of the backend, the execution times of the backend was measured for different input sizes of the *2D Convolution* benchmark. Figure 5.7 shows the the results. The chosen optimization level for the measurements was 07. The measurement consist of the execution times needed for the completion of the stages until the generation of the C code and when the input image sizes are 64x64, 96x96, 128x128 and 160x160. At input size of 96x96 it takes 50 seconds to execute the backend. By increasing this input size to 128x128 it takes the backend almost 2.8 minutes to be execute. An input image of 160x160 requires around 7 minutes to be processed

in the backend. Assuming this rate of growth between iterations is consistent it would require around 28 minutes to process a 256x256 input image, which is a long time for a modest pixel resolution. Thus, the current implementation of the approach is not very scalable for large input traces. Analyzing the time required for every stage it is clear that the reason for the large increase is the stage 2 processing. The time required for stages 4 to 7 do not change depending on the input size since stage 3 always compacts the input DFG to the same size. Stage 3 also does not take much longer for bigger inputs. Because stage 2 processing allows the stage 3 matching to be applied efficiently. It is stage 2 that requires substantial execution time. An output image of 96x96 pixels has 9216 outputs and 128x128 has 16384 outputs, i.e. almost twice as many outputs. As described in the previous chapter, the backend isolates every dataflow that generates an output. Each is leveled and then compared for common nodes. This implies that for a 128x128 image, the backend generates, levels and compares 16384 dataflows. A way to accelerate the backend and make the approach more scalable would be a more efficient stage 2 implementation. For example, the stage could be optimized to separate common nodes and unique nodes as the DFGs of the outputs are being generated.

5.4 Summary

This chapter presented the results obtained by the tool when applied to six benchmarks. The first section presented the benchmarks used to evaluate the tool. The tool is applied to the benchmarks with different possible configurations to check how they affect the resulting C code. The tool is compared with three different implementations for each benchmark. These implementations have different levels of optimizations so that one can compare the results with many different levels of expertise. The next section presented the speedups the tool obtained compared to these inputs, as well as the resource, latency and cycle time given through by Vivado HLS. The results the tool obtains show that for every input level it achieves better performance for every benchmark. The final section presents the execution times of the backend for different optimizations, as well as the scalability of the tool.

The results presented strongly show the usefulness of the approach and specifically of the framework. For every single benchmark, the framework achieved a restructured code that is on par or superior to the original C implementation, even when manually optimized with some directives. All C codes presented to show the approach were fully generated by the tool without manual intervention. Since the tool implements the optimizations automatically users would only require understating the configurations to obtain the optimized C code, and would not need a deep understanding of FPGAs and Vivado HLS. Our results show that optimizations that increase the ILP are the source of the biggest speedups. This increased ILP can be achieved through minimizing memory accesses or restructuring the arithmetic operations. For many of our benchmarks the memory accesses were a consistent bottleneck, and therefore the optimizations that handled the memory accesses tended to achieve the better results. The *filter subband* and *Autocorrelation* benchmarks also show that just applying the graph-based approach to folding and unfolding the input DFG,

can achieve improved results. These results are a strong motive to pursue further developments to the tool.

Chapter 6

Conclusion

This dissertation presents an approach to transform software code in order to be more suitable to high-level synthesis (HLS) tools. The approach is based on a dataflow graph (DFG) representation of the computations (at arithmetic, logical, operator level) currently obtained by executing the critical functions of the application previously added with instrumentation code. The approach relies on folding and unfolding graph operations and transforming the structure of the graph itself. The approach has been implemented in a framework able to fully restructure the code of critical application kernels. Although in the current work C code is considered as input, the approach has the potential to address different input programming languages via the inclusion of adequate instrumentation code. The framework consists of two stage, a frontend and a backend. The frontend generates DFGs from the execution trace through injection of instrumentation code to the original version. Currently the instrumentation code is inserted manually following certain rules to generate the appropriate DFG. The backend of the framework is capable of automatically restructuring the DFG and generate C code added with directives in a HLS-friendly way. The results, when targeting Xilinx FPGAs and using Vivado HLS, achieved are very promising. When compared with the original C codes, the C code generated by the framework outperforms them and significant speedups are achieved. The achieved C code is even comparable and in most cases better than manually optimized C code added with directives. When compared with the original unmodified C code the approach obtains implementations that are 30 to 100 times faster. When compared with C optimized with Vivado HLS directives the approach obtains implementations that are 2 to 15 times faster. However, the C code with directives generated by the framework can be always replicated by manual code transformations applied by experts. Thus, the approach can enable software developers to target efficient hardware accelerators using C code as input and typical HLS tools as backend, without requiring support of HLS, such as Vivado HLS, experts.

6.1 Future work

However the framework is in its initial phase and additional work is planned to improve it. Ongoing work is focused on additional DFG optimizations. An obvious advancement is the automation

of the insertion of the instrumentation code.

Future work will be focusing on the support to conditional constructs (if-else, switch), on more complex memory optimizations through analysis of the DFG, and on parameterized schemes to enable the representation of large execution traces in a DFG. Beyond supporting conditional constructs at the input code, another important feature is making the backend capable of creating new DFGs with those conditional statements. When conditional statements are simple cases, like comparisons between integers, then they can be very efficiently implemented with multiplexers, and even more efficient than having a more complex dataflow to circumvent the conditional statements. Additionally, even if the tool supports conditional statements at the input it needs to be able to optimize them. The statements might be unnecessary or they might be implemented in a manner that is non optimal.

Another essential focus of the future work is the interface of the kernel. HLS tools such as Vivado HLS have many different options of interfaces and these have a profound effect on the implementation. Improving the tool to implement these when appropriate or specified by the user is essential, as hardware/software interfaces can be difficult to address by software programmers.

Beyond adding more features to the tool, another important future work is increasing its flexibility, primarily in the way the tool folds the graphs. To handle more complex kernels and even applications the way the tool identifies patterns and creates loops should be more flexible, instead of just two distinct passes. It would also be interesting to use the matching algorithm to not just generate loops but also identify similar patterns that may not be implemented as loops, but could be optimized together. Increasing the flexibility is essential to expand the tool to handle large applications instead of just kernels.

Lastly, an improvement for the tool would be a better feedback system. It is important to inform the developer why certain optimizations fail, so that they might improve the code. The tool is limited by the information it is given. For example, the tool assumes any of the inputs variables can be any possible values. However if the programmer knows that the input is always going to have a specific value, then it would be better to define the input as a constant in the actual kernel, because hardware operations with constants are implemented differently than operation with variables. The tool could inform the user of this fact so that they might use constants in the code and allow for further optimizations.

All of these aspects could further improve the good results of the current tool. This clearly shows the growth potential of the graph-based approach chosen for this dissertation.

References

- [1] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13, 2013. doi:<https://doi.org/10.1016/j.jpdc.2012.04.003>.
- [2] Pong P Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. 2007.
- [3] C. Unsalan and B. Tar. *Digital System Design with FPGA: Implementation Using Verilog and VHDL*. McGraw-Hill Education, 2017. URL: <https://books.google.pt/books?id=VWYOvgAACAAJ>.
- [4] Steve Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007.
- [5] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 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, Oct 2016. doi:[10.1109/TCAD.2015.2513673](https://doi.org/10.1109/TCAD.2015.2513673).
- [6] Sanjay Churiwala and Sapan Garg. *Principles of VLSI RTL Design: A Practical Guide*. Springer Publishing Company, Incorporated, 1st edition, 2011.
- [7] Samir Palnitkar. *Verilog®Hdl: A Guide to Digital Design and Synthesis, Second Edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.
- [8] João M. P. Cardoso and Markus Weinhardt. *FPGAs for Software Programmers*, chapter 2, pages 23–47. Springer International Publishing, Cham, 2016. doi:[10.1007/978-3-319-26408-0_2](https://doi.org/10.1007/978-3-319-26408-0_2).
- [9] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. *Source-to-Source Optimization for HLS*, pages 137–163. Springer International Publishing, 2016. URL: https://doi.org/10.1007/978-3-319-26408-0_8, doi:[10.1007/978-3-319-26408-0_8](https://doi.org/10.1007/978-3-319-26408-0_8).
- [10] S. A. Edwards. The challenges of synthesizing hardware from C-Like languages. *IEEE Design Test of Computers*, 23(5):375–386, May 2006. doi:[10.1109/MDT.2006.134](https://doi.org/10.1109/MDT.2006.134).
- [11] Xilinx. Vivado design suite user guide: High level synthesis, 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug871-vivado-high-level-synthesis-tutorial.pdf.

- [12] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. Legup: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013. URL: <http://doi.acm.org/10.1145/2514740>, doi:10.1145/2514740.
- [13] 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*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM. URL: <http://doi.acm.org/10.1145/1950413.1950423>, doi:10.1145/1950413.1950423.
- [14] LLVM. The LLVM compiler infrastructure project, 2018. URL: <https://llvm.org>.
- [15] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. A. Najjar, and W. Bohm. An automated process for compiling dataflow graphs into reconfigurable hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):130–139, Feb 2001. doi:10.1109/92.920828.
- [16] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.
- [17] Ricardo Menotti, João M. P. Cardoso, Marcio M. Fernandes, and Eduardo Marques. LALP: A language to program custom FPGA-Based acceleration engines. *International Journal of Parallel Programming*, 40(3):262–289, Jun 2012. URL: <https://doi.org/10.1007/s10766-011-0187-0>, doi:10.1007/s10766-011-0187-0.
- [18] O. Mencer. ASC: a stream compiler for computing with fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1603–1617, Sept 2006. doi:10.1109/TCAD.2005.857377.
- [19] O. Mencer. PAM-Blox II: design and evaluation of c++ module generation for computing with fpgas. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 67–76, 2002. doi:10.1109/FPGA.2002.1106662.
- [20] C. Pilato and F. 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, Sept 2013. doi:10.1109/FPL.2013.6645550.
- [21] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. W. Janneck. High-level synthesis of dynamic dataflow programs on heterogeneous MPSoC platforms. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 227–234, July 2016. doi:10.1109/SAMOS.2016.7818352.
- [22] Maxeler Technologies. Maxcompiler white paper, 2017. <https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf>.
- [23] J. M. P. Cardoso, J. Teixeira, J. C. Alves, R. Nobre, P. C. Diniz, J. G. F. Coutinho, and W. Luk. Specifying compiler strategies for FPGA-based systems. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 192–199, April 2012. doi:10.1109/FCCM.2012.41.

- [24] Vasileios Tsoutsouras, Konstantina Koliogeorgi, Sotirios Xydis, and Dimitrios Soudris. An exploration framework for efficient high-level synthesis of support vector machines: Case study on ECG arrhythmia detection for Xilinx Zynq SoC. *Journal of Signal Processing Systems*, 88(2):127–147, Aug 2017. URL: <https://doi.org/10.1007/s11265-017-1230-1>, doi:10.1007/s11265-017-1230-1.
- [25] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000.
- [26] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 8(3):14:1–14:26, May 2015. URL: <http://doi.acm.org/10.1145/2629547>, doi:10.1145/2629547.
- [27] Shaoyi Cheng and John Wawrzynek. High level synthesis with a dataflow architectural template. *CoRR*, abs/1606.06451, 2016. URL: <http://arxiv.org/abs/1606.06451>, arXiv:1606.06451.
- [28] N. Voss, S. Girdlestone, O. Mencer, and G. Gaydadjiev. Automated dataflow graph merging. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 219–226, July 2016. doi:10.1109/SAMOS.2016.7818351.
- [29] Ashish Mishra, Mohit Agarwal, Abhijit Rameshwar Asati, and Kota Solomon Raju. Using graph isomorphism for mapping of data flow applications on reconfigurable computing systems. *Microprocessors and Microsystems*, 51:343 – 355, 2017. URL: <http://www.sciencedirect.com/science/article/pii/S0141933116304239>, doi: <https://doi.org/10.1016/j.micpro.2016.12.008>.
- [30] GraphViz. Graphviz- graph visualization software documentation, 2018. URL: <https://www.graphviz.org/documentation/>.
- [31] João M. P. Cardoso, Pedro C. Diniz, Zlatko Petrov, Koen Bertels, Michael Hübner, Hans van Someren, Fernando Gonçalves, José Gabriel F. de Coutinho, George A. Constantinides, Bryan Olivier, Wayne Luk, Juergen Becker, Georgi Kuzmanov, Florian Thoma, Lars Braun, Matthias Kühnle, Razvan Nane, Vlad Mihai Sima, Kamil Krátký, José Carlos Alves, and João Canas Ferreira. *REFLECT: Rendering FPGAs to Multi-core Embedded Computing*, pages 261–289. Springer New York, New York, NY, 2011. URL: https://doi.org/10.1007/978-1-4614-0061-5_11, doi:10.1007/978-1-4614-0061-5_11.
- [32] Texas Instrument. Tms320c6000 dsp library (dsplib), 2018. URL: <https://11vm.org>.
- [33] Corina G.Lee. Homepage of Corinna G. Lee, 2018. URL: <https://11vm.org>.

Appendix A

Benchmark C code

This appendix shows all the source codes for the benchmarks used in this dissertation.

```
void filter_subband_double_golden(double z[Nz],
double s[Ns], double m[Nm]) {
    double y[Ny];
    int i, j;
    for (i=0; i<Ny; i++) {
        y[i] = 0.0;
        for (j=0; j<(int) Nz/Ny; j++)
            y[i] += z[i+Ny*j];
    }

    for (i=0; i<Ns; i++) {
        s[i]=0.0;
        for (j=0; j<Ny; j++)
            s[i] += m[Ns*i+j] * y[j];
    }
}
```

Listing A.1: *filter subband* original source code

```
int DSP_dotprod_c(const short *x, const short *y, int nx)
{
    int sum = 0, i;
    for (i = 0; i < nx; i++)
        sum += x[i] * y[i];
    return sum;
}
```

Listing A.2: *dotproduct* original source code

```
void aut(short sc[N], short sd[M+N]) {
    for (i = 0; i < M; i++) {
        sum = 0;
```

```

    for (k = 0; k < N; k++) {
        sum += sd[k+M] * sd[k+M-i];
    }
    ac[i] = (sum >> 15);
}

```

Listing A.3: *Autocorrelation* original source code

```

void convolve2d(int input_image[N][N],
int kernel[K][K], int output_image[N][N]) {
    int i;
    int j;
    int c;
    int r;
    int normal_factor;
    int sum;
    int dead_rows;
    int dead_cols;
    dead_rows = K / 2;
    dead_cols = K / 2;
    normal_factor = 0;
    for (r = 0; r < K; r++) {
        for (c = 0; c < K; c++) {
            normal_factor += abs(kernel[r][c]);
        }
    }

    if (normal_factor == 0)
        normal_factor = 1;

    for (r = 0; r < N - K + 1; r++) {
        for (c = 0; c < N - K + 1; c++) {
            sum = 0;
            for (i = 0; i < K; i++) {
                for (j = 0; j < K; j++) {
                    sum += input_image[r+i][c+j] * kernel[i][j];
                }
            }
            output_image[r+dead_rows][c+dead_cols]
                = (sum / normal_factor);
        }
    }
}

```

Listing A.4: *ID fir* original source code

```
#define gamma
#define b
float sup_vectors[D_sv][N_sv];

void svm_predict(float test_vector[D_sv], float *sum){
    float diff;
    float norma;
    for(int i=0; i< N_sv;i++){
        for(int j=0; j<D_sv;j++){
            diff=test_vector[j] - sup_vectors[j][i];
            diff=diff*diff;
            norma=norma + diff;
        }
        *sum = *sum + (exp(-gamma*norma)*sv_coeff[i]);
        norma=0;
    }
    *sum= *sum-b;
}
```

Listing A.5: SVM original source code

Appendix B

Operation Mode

The backend of the framework has been implemented in Java and uses the Graphstream API.

Users indicate the configurations for the backend in a JSON file. The backend tool is executed in command line using:

```
java -jar backend.jar config.json
```

where the file "config.json" identifies the configuration file for a specific input example and optimization options. In this configuration file (see an example in Listing B.1), users must indicate the names and types of the inputs and outputs, the maximum sizes of the arrays involved and the name of the dot file with the input DFG. Users can choose to have no, medium or high folding. Other options include "arithmetic", "data_reuse" and "full_partitioning".

```
{
  "loadstores":4,
  "inputs":[
    "z[512]",
    "m[1024]"
  ],
  "input_types":[
    "double",
    "double"
  ],
  "outputs":"s[32]",
  "output_type": "double",
  "folding" :high,
  "arithmetic_optimiaztion" : true,
  "data_reuse" : false,
  "full_partitioning" : true,
  "graph" : "fsubba_graph.dot"
}
```

Listing B.1: JSON configuration file example for the *filter subband* benchmark