FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Acceleration of Applications with FPGA-Based Computing Machines: Code Restructuring

Tiago Lascasas dos Santos

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João M. P. Cardoso, PhD

Co-supervisor: João Bispo, PhD

July 31, 2020

Acceleration of Applications with FPGA-Based Computing Machines: Code Restructuring

Tiago Lascasas dos Santos

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. João Paulo de Castro Canas Ferreira, PhD
External Examiner: Dr. José Gabriel de Figueiredo Coutinho, PhD
Supervisor: Prof. João Manuel Paiva Cardoso, PhD

July 31, 2020

Abstract

Field-programmable gate arrays (FPGAs) can be used to accelerate performance-critical programs from a wide range of fields and still providing energy-efficient solutions. Programs written in high level languages, such as C and C++, can be compiled to FPGAs through High-level Synthesis (HLS). Although FPGAs benefit the most from parallel and data-streaming applications, efficient compilation to FPGAs is a problem for both tools and developers. Most applications do not follow these patterns, and extensive code restructuring and the use of HLS directives need to be applied to a program in order to take advantage of FPGAs. Code restructuring and the use of HLS directives often needs to be manually performed by an experienced developer, and as such there is a need to automate this process. This dissertation proposes a framework that automatically optimizes C code via directives, using a source-to-source compiler on a stage prior to HLS. This optimization is primarily applied by strategies that select, configure and insert directives on the code to be input to an HLS tool, e.g., Vivado HLS, in order to synthesize more efficient hardware accelerators. Those strategies rely on very simple but effective heuristics, which use a small set of properties extracted from the control/dataflow graphs generated from the computations being compiled. The framework is evaluated using a wide variety of input source code, and the results show that the framework manages to achieve efficient speedups across all benchmarks when compared to their unoptimized versions, while maintaining a low resource usage in most cases. The framework is also compared to code optimized manually with directives, and the experiments show that it achieves similar results.

Keywords: FPGAs, Code Optimization, High-level Synthesis, Source-to-Source Compiler

Resumo

Os dispositivos lógicos programáveis conhecidos por FPGAs podem ser usados para acelerar programas com requisitos de performance exigentes e provenientes de uma vasta gama de domínios, com a vantagem adicional de manter uma alta eficiência energética. Programas escritos em linguagens de alto nível, como C e C++, podem ser compilados para FPGAs através da Síntese de Alto Nível (HLS). Apesar de os FPGAs tirarem melhor partido de aplicações com paralelismo e de fluxo de dados, a compilação eficiente para estas plataformas é um problema tanto para as ferramentas como para os programadores. Visto que maior parte das aplicações não seguem estes padrões, uma reestruturação de código extensiva e a aplicação de diretivas de HLS têm de ser aplicadas a um programa de modo a se poder tirar partido do FPGA. A reestruturação de código e a inserção de diretivas, normalmente, têm de ser efetuadas manualmente por um programador com experiência e, portanto, existe a necessidade de se automatizar este processo. Esta dissertação propõe uma ferramenta que otimiza código C automaticamente num compilador *source-to-source* numa fase anterior à HLS. Esta otimização é, maioritariamente, aplicada através de estratégias que selecionam, configuram e inserem diretivas no código que serve de input a uma ferramenta de HLS, p. ex. Vivado HLS, de modo a sintetizar aceleradores mais eficientes. A ferramenta é avaliada usando uma grande variedade de códigos fonte, e os resultados mostram que a ferramenta consegue atingir ganhos eficientes de latência quando comparados com as suas versões não otimizadas, mantendo uma utilização de recursos baixa na maior parte dos casos. A ferramenta também é comparada com código otimizado manualmente com diretivas, e as experiências mostram que os resultados são semelhantes aos obtidos com a inserção manual de diretivas.

Palavras-chave: FPGAs, Otimização de Código, Síntese de Alto Nível, Compiladores fonte-para-fonte

Acknowledgments

I would, first and foremost, like to express my absolute gratitude to my supervisor, João Cardoso, for all the constant guidance, availability, corrections and feedback provided during this dissertation. It was an honor to be able to work with you. A very special thanks, too, to my co-supervisor, João Bispo, for always being there to receive the oddest of questions at the oddest of hours, and for always going the extra mile in his answers. My thanks, as well, to the other folks in the SPeCS lab: to Renato Campos, for the constant exchange of ideas and for always being there to help me out; to Luís Noites, for those discussions during the early days of this dissertation; to Pedro Pinto, for laying the groundwork that I would then build upon; and to the rest of the lab members, for creating a stimulating and pleasant environment to work in. This whole pandemic ordeal made it quite difficult to feel immersed in a laboratorial environment, but regardless, I am very glad to have been able to work in this organization. A special thanks, too, to Afonso Ferreira, whose previous advancements on this topic are incredibly relevant to my work, and needless to say, inspired it. I also acknowledge the academic program of Xilinx, Inc. for providing the lab with FPGA boards and licenses of the tools.

As for my family, I would like to thank my parents, Francisco and Diamantina, for the endless patience they've shown to me during this endeavor. I know that hearing me type on the keyboard deep into the night was a real nuisance. I also want to thank my brother, Gonçalo, who was always there to cheer me up. I look forward to read your own dissertation in a few years' time. Finally, I would like to dedicate this dissertation to my grandmothers, Delfina and Emília. The other day I found my kindergarten yearbook, in which you both left me messages for the future. Your eagerness for me to find success radiated in them, and so I'd like to acknowledge that by dedicating this work to you.

Tiago Lascasas dos Santos

”Always an even trade”

Steven Erikson

Contents

List of Figures

List of Tables

Abbreviations

AOP	Aspect-Oriented Programming
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
BRAM	Block Random Access Memory
CDFG	Control and Data Flow Graph
CFG	Control Flow Graph
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DFG	Data Flow Graph
DSE	Design Space Exploration
DSL	Domain-Specific Language
DSP	Digital Signal Processor
ECP	Estimated Clock Period
FIFO	First-In, First-Out (referring to a queue)
FP	Floating-point
FPGA	Field Programmable Gate Array
FIR	Finite Impulse Response
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
II	Initiation Interval
IR	Intermediate Representation
kNN	k-Nearest Neighbours
MCF	Maximum Clock Frequency
LUT	Lookup Table
MIR	Metasemantic Intermediate Representation Graph
OS	Operating System
RAM	Random Access Memory
RTL	Register-transfer Level
SCC	Strongly-connected Component
SoC	System on a chip
SVM	Support Vector Machine

Chapter 1

Introduction

Across the industry, there is a wide variety of applications from diverse fields, such as digital signal processing, networking, finance, bioinformatics and computer vision, among many others [1], that are always seeking to improve performance while keeping energy costs low. Standard CPUs are often not enough to fulfil these needs, and in order to achieve significant gains there is a need to develop hybrid applications that execute partially on a CPU, and partially on hardware accelerators. This kind of heterogeneous computing is done by identifying performance critical code on an application, and delegating that code to an hardware accelerator while the rest of the code keeps running on a standard CPU [2] [3].

The most common kind of hardware accelerators that are able to run more than a single, hardcoded program are Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Application-specific Integrated Circuits (ASICs) are also popular accelerators, but they are designed to execute a single program and therefore are not appropriate for reconfigurable heterogeneous computing [4]. FPGAs can have their internal hardware reconfigured in order to implement different algorithms, which makes them an ideal candidate to be used in heterogeneous computing. While GPUs are often better at executing complex and extensively parallel applications, FPGAs hold an advantage over GPUs when it comes to executing code segments characterized by enabling data parallelism and by having relatively simple data objects (e.g., no dynamically allocated objects and limited use of pointers) and arithmetic operations [3].

FPGAs can be integrated into other systems in many different ways. FPGAs can come in the form of PCI Express cards, such as the Intel FPGA Programmable Acceleration Cards [5] and the Xilinx Alveo Cards [6], which can be inserted on a typical desktop computer or server. Others may be part of a system-on-a-chip (SoC) embedded on a development board and with an integrated CPU. One such example is the Xilinx Zynq SoC [7], which can be found embedded on a ZedBoard Development Kit [8]. Finally, FPGAs can also be integrated on distributed systems running on the cloud. Current cloud service providers, such as Amazon Web Services (AWS) and

Microsoft Azure, already provide cloud-based FPGA platforms on demand, with native integration to the other cloud services provided by them [9] [10].

1.1 Objectives

1.1.1 Problem Description

Despite the advantages offered by FPGAs, developing an application with these platforms in mind is not an easy task for a software developer. FPGAs are often programmed using an Hardware Description Language (HDL), such as VHDL [11] or Verilog [12], in a process that is often time-consuming and error-prone [13]. An alternative to using HDLs is programming the code segments in a typical language used in software development, such as C or C++, which are often the languages used for the rest of the application running on the CPU. This facilitates the developer's workflow, since the compilation for the FPGA can, in some cases, be abstracted.

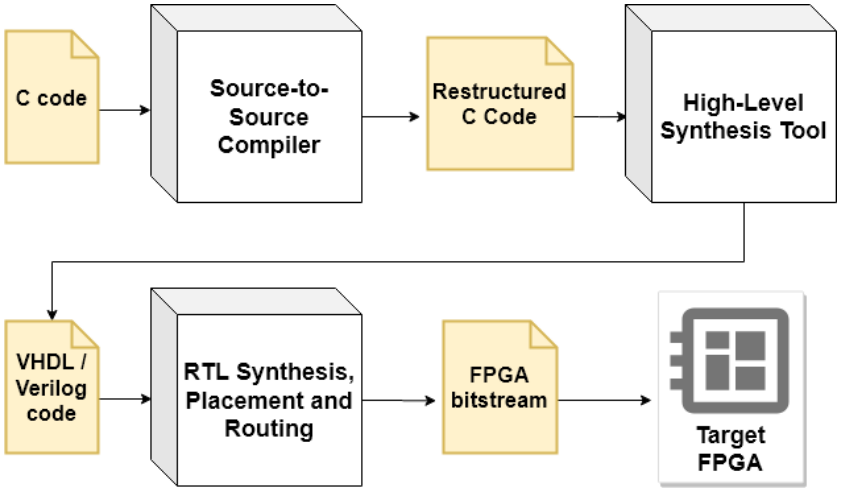

Figure 1.1: Example of a possible compilation pipeline for a C program targeting an FPGA

However, compilation of a language like C to an FPGA is a complex process, which involves the compilation of the language to an HDL in a process known as High Level Synthesis (HLS) [14]. The resulting HDL is often far from optimal, since a typical C program does not have the characteristics that can be exploited by an FPGA, and as such it is hard to take full advantage of its resources and advantages. It is, then, necessary to restructure the C code on a stage prior to the HLS in order to produce a more appropriate hardware description and subsequent FPGA implementation. Figure 1.1 exemplifies this workflow: a C application is first restructured on a source-to-source compiler, and then synthesized by an HLS tool in order to produce an HDL model that can be further automatically translated into a bitstream used to configure the FPGA.

This dissertation, then, proposes a framework that attempts to alleviate the burden imposed on a developer by applying code restructuring procedures on a source-to-source compiler, using minimal to no user input besides the source files and the indication about which functions should be synthesized.

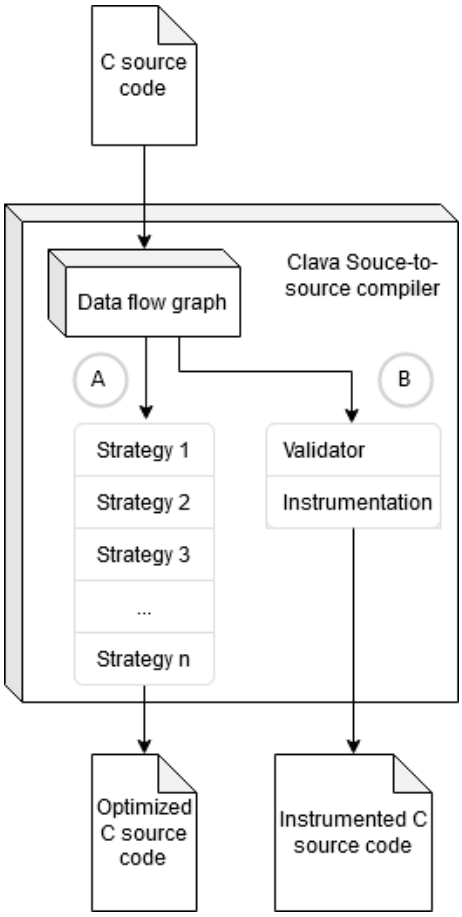

Figure 1.2: Approach adopted by the proposed framework

1.1.2 Proposed Solution

The proposed solution is integrated in the Clava C/C++ to C/C++ compiler, developed by the SPeCS group [15]. The developed solution is based on the construction of a data flow graph, which can be analyzed in order for code restructuring procedures to be applied. This is exemplified by the flow A in Figure 1.2. The code restructuring strategies are based on selecting, configuring and inserting directives that will then be used by the HLS tool to synthesize a more efficient hardware model, as well as some simple code transformations based on the compile-time optimization of mathematical functions. These strategies are guided by heuristics, and aim to reduce the overall latency of the targeted functions while keeping resource usage within the budget provided by the targeted FPGA. Many existing approaches also focus on optimizing a single kernel, or a small set of similar kernels. This approach, comparatively, is more generic, as it is suitable for different kinds of kernels (e.g., machine learning kernels, filters and matrix manipulation kernels, among others). An integration with another existing code restructuring tool [16], also developed by SPeCS, is also considered as an alternative work flow. In this case, the framework is used to automate the initial step of that preexisting approach. This secondary use case is exemplified in flow B of Figure 1.2. The results are evaluated by comparing the latency and resource usage

of restructured code to the unoptimized version of that code and to versions of code restructured manually by an experienced developer. The main contributions are:

- A new directive-based C code optimization framework for HLS integrated on a source-to-source compiler;
- New heuristics to guide the code optimization strategies applied by the framework;
- Automatization of the validation and instrumentation steps of the existing code restructuring approach proposed by Ferreira and Cardoso [16];
- A near-instantaneous and automatic workflow that can compete against manual code optimization performed by an experienced developer.

1.2 Structure of the Dissertation

This dissertation is structured in 6 chapters. Chapter 1 gives an overview of the problem and its context, as well as the general objectives that the proposed solution aims to achieve. Chapter 2 presents the results of a literature review that details the latest advancements in code restructuring for FPGAs, plus an overview of modern HLS tools. Chapter 3 presents the code analysis framework, which is used to create graph-based representations of source code suitable for analysis. Chapter 4 focuses on the code restructuring strategies that are applied to the code, based on the previously obtained graph. Chapter 5 details the experimental results of the framework, with the conclusion coming last in Chapter 6.

Chapter 2

Related Work

Efficient compilation of applications to FPGAs depends on both the optimizations applied to the code before it is compiled, as well as the optimizations introduced by the High-Level Synthesis (HLS) tools. This chapter presents an overview about the current HLS tools, followed by the presentation of related work regarding seven distinct approaches to code restructuring for FPGAs.

2.1 High-Level Synthesis Tools

Multiple HLS tools are available today on both the industrial and academic fields [17] [18]. A general overview of the amount and type of HLS tools is presented, followed by a more in-depth description of two HLS compilers: one commercial and one academic.

2.1.1 Overview

A survey [19] from 2016 identifies and describes 33 HLS tools by enumerating their properties. Some of the properties used to describe a tool were:

- Whether the tool is in active use or is abandoned. Around half of the tools were identified as being actively used;
- The type of license, which can be commercial or academic (with a split of 40%/60% between them);
- The input language. Most tools accept C or C++ as the input, as well as subsets of those languages and DSLs. Other high-level languages, such as Java, MATLAB and C#, are also accepted by at least one tool each;
- The output language. HLS tools usually output a HDL, with Verilog [12] and VHDL [11] being the most common. Many tools are able to output both. SystemC [20] can also be

generated by some tools, but it is not as common. Some tools may also automatically synthesize the hardware bitstream;

- The domain of the tool. Some tools may be generic and synthesize code from any domain, while others may focus on synthesizing code from a specific domain, such as image processing or streaming applications;
- A tool can be further characterized by its release year, benchmark capabilities and support for floating point and fixed point arithmetic;

The optimization mechanisms applied by the HLS tools were also gathered and grouped in the following categories:

- *Operation chaining* - optimization that chains two combinational operations in order to fit in a single clock cycle;
- *Bit-width Analysis and Optimization* - attempts to transform the datapaths into using bit-widths different from the ones declared for certain variables, in order to create data paths that better reflect the actual size requirements of the data. This provides gains on both the area, performance and power consumption;
- *Memory Space Allocation* - tries to distribute the available BRAM modules with the purpose of enabling the parallel access to different BRAM modules on a single cycle;
- *Loop Optimizations* - introduces multiple loop optimization restructurings, such as loop pipelining, in order to enable loop-level parallelism;
- *Hardware Resource Libraries* - identifies operations on the code that can be implemented by a set of pre-built hardware resources;
- *Speculation and Code Motion* - on data-driven applications, it may be useful to speculatively execute code before deciding whether the code should have actually been executed;
- *Spatial Parallelism Exploitation* - enable instruction and loop-level parallelism by executing code segments in parallel, with particular care taken to preserve the data dependencies between them. This is often achieved by HLS tools that allow the compilation of *pthreads*, OpenMP and OpenCL;
- *If-Conversion* - converts if-statements so that the code they guard is only executed when it evaluates to true. This allows for disjoint execution paths controlled by if-statements to be scheduled in parallel.

Finally, four HLS tools were selected to be benchmarked: three academic (Bambu [21], DWARV [22] and LegUp [18]) and one commercial tool whose identity was not specified. For each tool, they 17 different codebase suites, both using the default settings and the optimization

options. Execution times were measured in terms of clock cycles latency, maximum clock frequency and wall-clock time, while the FPGA resource consumption was measured by counting the number of LUTs, BRAMs and DSPs required by each bitstream. These results showed that no tool proved to be better than the others across all benchmarks, and that there are no drastic differences between commercial and academic tools. However, a point is made that commercial tools have extra advantages over the academic ones, such as more variety of input languages and more robustness.

2.1.2 Vivado HLS

Vivado HLS [17] is a commercial HLS tool by Xilinx. It provides an environment similar to those of regular software development by providing both command-line tools and an IDE, with the only difference being that it targets FPGAs rather than the usual CPUs. It accepts C and C++ as the input, and can output VHDL [11], Verilog [12] and SystemC [20]. The developer may introduce directives via pragmas in the code in order to enable optimizations to be performed by the compiler, as well as to establish the kind of interface for the inputs and outputs of the program. Some examples of possible optimizations enabled by the use of directives are kernel optimization, function inlining, loop unrolling, array partition and pipelining [23]. The development environment also provides a test bench that aids in proving the functional correctness of the synthesized solution, as well as simulation and code coverage tools for further testing.

2.1.3 LegUp

Canis et al. [18] propose the open-source HLS compiler LegUp. This compiler accepts C as the input, and targets a hybrid 32-bit MIPS CPU and FPGA system. Based on LLVM, it is structured in order to allow for the easy implementation of new HLS algorithms within the compiler itself, and its open-source nature is stated as being a differentiator when compared to the other tools that came before it, as well as the full support for certain C language constructions, such as structs and pointer arithmetic. Due to the hybrid nature of its target, the compiler is able to automatically identify regions of the code that could benefit from hardware acceleration, and generate HDL for those regions. The LegUp compiler also led to the creation of a company, LegUp Computing, which focuses on its further development and commercialization [24].

2.2 State-of-the-art Code Restructuring Strategies

This subsection describes seven state-of-the-art code restructuring approaches for code targeting an FPGA through High-Level Synthesis.

2.2.1 Design Space Exploration (DSE)

Tsoutsouras et al. [25] focus on improving the prediction phase of a Support Vector Machine, as presented in Listing 2.1, by targeting it to an FPGA. The optimal design consists of two stages.

The first stage restructures the code before the HLS process using two strategies. The first strategy identifies regions in the code that can be parallelized, such as the calculation of the euclidean distance of each support vector of the SVM with the test vector. The code is modified by partitioning the support vectors array into multiple smaller vectors, and then by using HLS directives on each partition to enable loop-level parallelism. Measurements indicate that the speedup achieved correlates directly to the amount of partitions (e.g., 2×speedup for a partition factor of 2 and a 16×speedup for a partition factor of 16). The second strategy unrolls a loop-based arithmetic calculation into a tree-like structure in order to achieve better results than the loop unroll directive of Vivado HLS.

```
1  const float sv_coef[N_sv];
2  const float sup_vectors[D_sv][N_sv];
3
4  void SVM_predict (float test_vector[D_sv], int * y) {
5      loop_i: for (i=0; i<N_sv; i++){
6          loop_j: for (j=0; j<D_sv; j++) {
7              diff = test_vector[j] - sup_vectors[j][i];
8              norma = norma + diff * diff;
9          }
10         sum = sum + exp(-gamma * norma) * sv_coef[i];
11         norma = 0;
12     }
13     sum = sum - b;
14     if (sum < 0) *y = -1
15     else *y = 1;
16 }
```

Listing 2.1: SVM original prediction code

The second stage performs design space exploration over 4 Vivado HLS directives (Loop pipeline, Loop unroll, Array partition and Array reshape), all their different parameters and all the code regions in which they can be applied. Some pruning is first done to remove designs in which directives are applied to regions that would not benefit from them, as well as designs that would be functionally equivalent to other designs. Further pruning is done by applying 3 pruning guidelines that relate the unrolling factor of a loop to the reshaping and partition factors of the code referenced by that loop. These pruning directives were successfully validated by calculating the full design space and the pruned design space, and then performing a Pareto analysis that considered the trade-off between delay and area utilization. The pruning guidelines reduced the design space by 97.44%, giving a reduced number of candidate designs. An optimization algorithm that takes into account the FPGA resources and delay can then be used to determine the final design.

The experimental evaluation was performed using a Zedboard Zynq Development Kit [8], which includes a xc7z020clg484-1 Xilinx FPGA [7], and an SVM trained with data to detect arrhythmia from ECG data. Multiple experiments were conducted. The first evaluated the proposed DSE methodology by using two different optimizers for the final step, as well as designs from the

full design space. This revealed that the pruned designs were very close to the optimal area-delay values. The second evaluation addresses the validation of this method when used for other SVM models with different scales, and it was successfully validated for SVMs with support vector sizes up to 100000 and up to 1000 features. Finally, the execution latency of the generated design was measured against standard CPUs using the ECG model, which revealed an execution latency gain of 98.78%.

2.2.2 Ordered Loop Transformations

Li et al. [26] propose an approach based on applying four code transformations on a predetermined order in order to improve real-time image processing algorithms. Firstly, an optimization based on function inlining is applied by determining the hierarchy of a set of function calls and then flattening that hierarchy into a single entity by injecting, recursively, the code of the functions in place of their respective calls. After the inlining is performed, all loop bodies are inside the same function. These loops are executed sequentially, even when they are independent of each other. Thus, merging the multiple loop bodies into being under a single loop is necessary. This is done by, firstly, converting all nested loops into simple loops. Then, all simple loops are merged into a single one. The control over which of the merged loop bodies should be executed on a given iteration is controlled through if-statements.

The program is then subject to a number of symbolic expression manipulations. These manipulations attempt to simplify certain arithmetic and boolean operations, and consist of the following:

- *Folding* - simplifies expressions with constant literals, e.g., $1+3x-4x+2$ becomes $3-x$;
- *Division* - simplifies expressions representing a division by precalculating the divisions between constants, e.g., $(x*2)/(y*10)$ becomes $x/(y*5)$;
- *Short-circuit evaluation* - identifies if some expressions can be reduced to 0, replacing them by that value;
- *Normalization* - moves the position of the variables and constants of a conditional operator in order to have variables on only the left side, and constants only on the right side, e.g., $1+x<y+2$ becomes $x-y<1$;
- *Segmentation* - Splits a long arithmetic expression into multiple shorter ones, e.g., $x+y+z+t$ becomes $temp1 = x+y$, $temp2 = z+t$ and $temp1+temp2$.

Finally, loop unwinding [27] is applied to the remaining loop. This has the potential of providing an acceleration factor of 2^n for an unrolling factor of n, but the actual value is sometimes lower due to delays introduced between iterations, which need to share the same top interface. Concern is also raised about the area of the FPGA used by this unrolling, since a high unrolling factor may require resources that fall outside the target FPGA's capabilities. Listing 2.2 shows an unoptimized code sample with a sequence of nested loops, and Listing 2.3 shows the result of the loop transformations applied to the former code sample.

```
1
2  void example()
3  {
4      //Loop 1
5      for (int i=0; i<N; i++)
6      {
7          //Body of Loop 1
8      }
9
10     //Loop 2
11     for (int i=0; i<N; i++)
12     {
13         //Body of loop 2
14
15         //Loop 2.1
16         for (int j=0; j<N; j++)
17         {
18             //Body of loop 2.1
19         }
20     }
21
22     //Loop 3:
23     for (int i=0; i<N; i++)
24     {
25         //Body of loop 3
26     }
27 }
```

Listing 2.2: Original code before Loop Manipulation

```
1
2  void example()
3  {
4      //Merged loops 1, 2 and 3
5      for (int i=0; i<N*N; i++)
6      {
7          if (i >= 0 && i < N)
8          {
9              //Body of loop 1
10             //Body of loop 2
11             //Body of loop 3
12         }
13         //Body of loop 2.1
14     }
15 }
```

Listing 2.3: Code after applying Loop Manipulation

This optimization process was evaluated by using four different algorithms used in real-time image processing: a 3×3 filter for RGB images, matrix product, image segmentation and stereo matching. Different integer data types for the input matrices are also used. The performance of this method is compared to those of the versions optimized using two other optimizers: Vivado HLS directives and the polyhedral optimizer PolyComp [28]. All three versions are synthesized using Vivado HLS and the target FPGA was a Xilinx xc7a200tfbg676-2 [7]. The proposed solution outperforms the other two implementations across all benches, with an average speedup of 106.54×when compared to an unoptimized version, while Vivado HLS and PolyComp had average speedups of only 22.19×and 19.01×, respectively. It is also observed that the first three optimizations, by themselves, do not effectively improve the design. Most of the performance gains comes from the loop unroll, which is the last transformation applied. However, the previous steps are still necessary, as the simplifications they introduce allow for a more efficient loop unrolling transformation.

2.2.3 Generating Optimized Code From Tracing-based Data Flow Graphs

Ferreira and Cardoso [16] propose a restructuring of code based on a frontend and a backend. The frontend, builds a direct acyclic dataflow graph from instrumentation data gathered through program execution tracing. This DFG contains nodes of three types that represent how the input data is changed throughout the execution of the program. These types are variables, constants and operations. It was designed to be as simple as possible in order to be easily ported to other languages besides C.

The backend consists on seven stages that focus on analyzing and optimizing the DFG obtained on the previous step. Stage 1 prunes the graph by removing unnecessary nodes, and by identifying patterns that can be folded. Stage 2 isolates all the dataflows that lead to each of the different outputs, while Stage 3 tries to merge similar dataflows into a single dataflow with a loop. Stage 4 tries to identify pipelining opportunities by by choosing the variable with the largest number of writes, while stage 5 applies two types of optimizations based on arithmetic operations and memory accesses. On Stage 6, the loops generated on stages 3 and 4 are unfolded. Finally, Stage 7 generates C code, together with the appropriate HLS directives. An example of a function restructured by this process is shown in Listing 2.5, while the original code is shown in Listing 2.4.

```
1  #define Nz 512
2  #define Ns 32
3  #define Nm 1024
4
5  void filter_subband (double z[Nz], double s[Ns], double m[Nm]) {
6      double y[Ny];
7      int i, j;
8      for (i = 0; i < Ny; i++){
9          y[i] = 0.0;
10         for (j = 0; j < (int)Nz / Ny; j++)
11             y[i] += z[i + Ny * j];
12     }
13     for (i=0; i < Ns; i++){
14         s [i] = 0.0;
15         for (j = 0; j < Ny ; j++)
16             s [i] += m[Ns * i + j ] * y[j];
17     }
18 }
```

Listing 2.4: Original source code for a Filter subband

The framework is evaluated by synthesizing code with Vivado HLS and by targeting a Xilinx Artix-7 FPGA (xc7z020clg484-1) [7]. Multiple code suites were used, with algorithms taken from DSPLIB [29], UTDSP Benchmark Suite [30] and an MPEG audio encoder [31], and the optimized versions are compared against versions that are optimized manually using HLS directives. The framework allows for 8 distinct optimization levels, and the speedup for levels 5 through 8 was

also measured. With each level, speedup increases across all algorithms, except for a few outliers which start to have no gains after a certain level. All algorithms have significant speedup gains when compared to the manually optimized versions.

```
1  void filter_subband_pipe(double z[512], double s[32], double m[1024]) {
2      #pragma HLS array_partition
3      variable = s cyclic factor=16 dim= 1
4      #pragma HLS array_partition
5      variable = z cyclic factor=16 dim= 1
6      #pragma HLS array_partition
7      variable = m cyclic factor=64 dim= 1
8      s[0] = 0;
9      ...
10     s[31] = 0;
11     for (int i = 0; i < 64; i = i + 4){
12     #pragma HLS pipeline
13         part11 = z[i + 320] + z[i + 256];
14         part12 = z[i + 321] + z[i + 257];
15         ...
16         y0_a20 = final_part3;
17         y0_a30 = final_part4;
18         for (int j =0; j < 32; j=j+1){
19             temp1 = m[(32) * j+i] * y0 ;
20             ..
21             temp4 = m[(32) * j+i] * y0_a30;
22             partial_in1 = temp1 + temp2;
23             ...
24             final_partin = part_in3 + part_in4;
25             s[j] = s[j] + final_partin;
26         }
27     }
28 }
```

Listing 2.5: Abridged version of the optimized source code for a Filter subband

2.2.4 Conversion of Code Into a Data Flow Engine

Cheng and Wawrzynek [32] suggest the mapping of a function into a multi-stage data flow engine based on the data flow architectural template. Each stage consists of a module that performs an operation every time an input is ready. These modules can have different granularity: on one end of the spectrum, we have a single module that spans the entire function, which would be equivalent to an unoptimized version. On the other end, we have a separate module for each individual operation, but this would incur in a very high area overhead.

The mapping is performed by, first, building a Control and Data Flow Graph (CDFG), and then partitioning it by cutting off dependency edges and mapping the nodes to different stages in

the pipeline. Strongly-connected components (SCC), which are components formed from circular dependencies in the CDFG, are determined, and are converted into new nodes that are then added to the CDFG. Then, the stages of the pipeline are determined whenever a SCC or a memory operation is encountered. Communication between the different stages, at the FPGA's level, is done through FIFO queues.

A series of potential optimizations are then considered. The first concerns itself with the trade-off between computation and communication. Since FIFOs incur in a significant area cost on the FPGA, it may make some sense to duplicate a computation rather than creating a new FIFO between two modules. Another optimization is about the creation of a new stage after a memory operation is detected, which causes all data requesters to decouple themselves from the module that issued the memory request. This allows for the access to sequential memory locations to be done in a burst access, which improves the efficiency of memory accesses.

This framework is implemented using LLVM's IR, which allows for the creation of a CDFG, and Vivado HLS is used for the synthesis. Experimental evaluation is done using 4 benchmarks, with the targeted FPGA being a Xilinx Zynq-7000 XC7Z020 [7] in a ZedBoard [8]. The system provides two accelerators to access the main memory subsystem: an accelerator coherence port (ACP) and a high performance port (HP). Benchmarks are analyzed for all 4 combinations of these accelerators as well. This development kit also has an ARM CPU, which is used to measure a baseline for each benchmark. Synthesis using an unoptimized version is also performed for comparison. The optimized version performed the same or better than the unoptimized version in all benchmarks, with an average speedup of 5.6×. It also has better performance than the ARM baseline on three out of the four benchmarks.

2.2.5 Abstractions for Targeting Different HLS Tools

Özkan et al. [33] address the problem of different HLS tools using different vendor-specific directives and code annotation methods, which hinder code portability. They propose AnyHLS, a library of abstractions that aim to encapsulate the functional behaviour of typical HLS directives, and whose backend can translate those functionalities into a format supported by the HLS tool and FPGA being targeted.

AnyHLS is implemented using AnyDSL [34], which is a framework that allows for the creation of domain-specific libraries. AnyDSL provides a language called Impala, which allows for the partial evaluation of its code at compile-time. This language is used to implement the code transformations commonly provided by directives in the form of abstractions. One such type of abstractions comprise of loop transformations, such as loop unrolling and pipelining, which are also parameterizable. Another type of abstraction pertains to the way memory is modelled in terms of data type sizes and access properties, including the kind of interface that should be established between an FPGA and a CPU (e.g., mapping to registers or using a data stream). Other abstractions, such as finite state machines, reductions and multiplexers are also provided.

These abstractions are then used to create a library for image processing, which offers a series of domain-specific higher level abstractions and optimizations. One such optimization is the

vectorization of input pixels (i.e., the mapping of input pixels to multiple vectors that are processed in parallel). The higher-level abstractions can, once again, be divided in memory and loop abstractions. The memory abstractions relate to connecting multiple image processing kernels through data streams, FIFO queues for the image lines to be processed and sliding windows with a memory-efficient updating method. The loop abstractions relate to the way the image is analyzed and modified. Different strategies are employed for when the processing is done on a pixel-by-pixel basis, such as a color transformation (point operators), and for when the processing is done over a region of the image, such as applying a Gaussian blur (local operators). Point operators produce, for each input pixel, an equivalent output pixel, and thus can take advantage of the aforementioned vectorization procedure to parallelize the processing of each pixel. Local operators go through an image and process a region in each iteration, and thus can take advantage of the efficient abstractions provided by the image line FIFOs and sliding window implementations.

The proposed abstraction library can target both Xilinx's Vivado HLS [17] and Intel's FPGA SDK for OpenCL [35]. The evaluation of the results is performed for both HLS compilers, and the target hardware is a Zynq XC7Z020 FPGA for the Vivado HLS design, and a Cyclone V GT 5CGTD9 FPGA for Intel's OpenCL SDK. The results are also compared against other two state-of-the-art DSL-based HLS tools, Halide-HLS [36] and Hipacc [37]. The benchmark suite consists of 7 image processing applications, such as a series of filters, a Harris corner detector and a Sobel edge detector. Results are measured in terms of hardware resources and throughput, measured in MB/s. Some optimizations, such as vectorization and streaming, are also evaluated individually. The final results show that AnyHLS can achieve similar results to the ones obtainable by the other tools, with some improvements when it comes to multi-kernel applications, such as the Harris corner detector.

2.2.6 Balancing the Latency, Area and Accuracy Trade-off

Gao, Wickerson and Constantinides [38] propose SOAP3, a tool that attempts to help a developer solve the trade-off between the latency and area gains of a synthesized FPGA design and its accuracy regarding floating-point accuracy and the enforcement of inter-iteration dependencies (e.g., a floating-point counter that is modified on each iteration). In order to trade-off between these three metrics, SOAP3 produces a set of restructured programs on the Pareto frontier (i.e., a set comprised of all programs P for which SOAP3 did not manage to find another program P' that improves P on all metrics).

The proposed framework starts by converting the input C code into a metasemantic intermediate representation graph (MIR), that attempts to strip the program of most of its structure, mantaining only the effects it should have. Then, the MIR is partitioned into several sub-MIRs, which are then optimized individually. The optimized sub-MIRs are then merged with each other using different combinations, and the ones on the Pareto frontier are chosen. Each individual sub-MIR is optimized using three kinds of transformations. The first kind is focused on arithmetic rules, which attempt to simplify and balance arithmetic expressions. The second kind is about

control flow rules, of which partial loop unrolling is the only one applied. Finally, there are transformations that try to reduce the number of accesses to a variable, such as eliminating multiple reads and multiple writes to the same variable, avoiding reading a variable after it was written to and reordering independent accesses to the same variable.

The tool is evaluated using 12 benchmarks, extracted from the Livermore Loops suite [39] and PolyBench [40]. The HLS compiler used is Vivado HLS targeting a Xilinx Virtex7 FPGA. The results show that the tool can output restructured programs that, when synthesized, offer up to 12×more speedup and and a reduction of 7×of round-off errors, with an increase of resource usage of up to 4×.

2.2.7 Use of Aspect-Oriented Programming

Cardoso et al. [41] propose the use of aspect-oriented programming (AOP) to apply code transformations on high-performance embedded systems, of which FPGAs are part of. They propose the aspect-oriented language LARA, which allows for a code restructuring strategy, called an aspect, to be defined on a separate file, which is then weaved into the application source code by the compiler. The usage of LARA for the particular case of FPGA compilation is further expanded upon in another article by Cardoso et al. [42].

There are multiple advantages to this approach. The application's source code does not need to be manually annotated, and restructuring strategies more complex than simple HLS directives can be specified. There is also the possibility of reusing the same strategy, which may be specific for a single hardware platform, to be applied on different applications. Multiple strategies can also be applied to a single application, and multiple versions of an application may be weaved by using multiple strategies. Finally, a strategy may be configured by different parameters, which lead to different designs. These parameter-dependent designs may be a starting point for further design space exploration. The different combinations are specified on Figure 2.1, which is based on a similar picture from the same article: approach A applies one aspect to a program; approach B applies multiple aspects to a program, producing multiple versions of that program; approach C applies the same aspect to multiple programs, producing one version of each program; approach D applies one aspect to one program, which, according to user-provided parameters, can produce multiple versions of that program. The biggest disadvantage of this approach is that it still requires the developer to manually develop the code restructuring strategies, and certain strategies may not be possible to codify due to limitations in the LARA specification and on the compiler implementation.

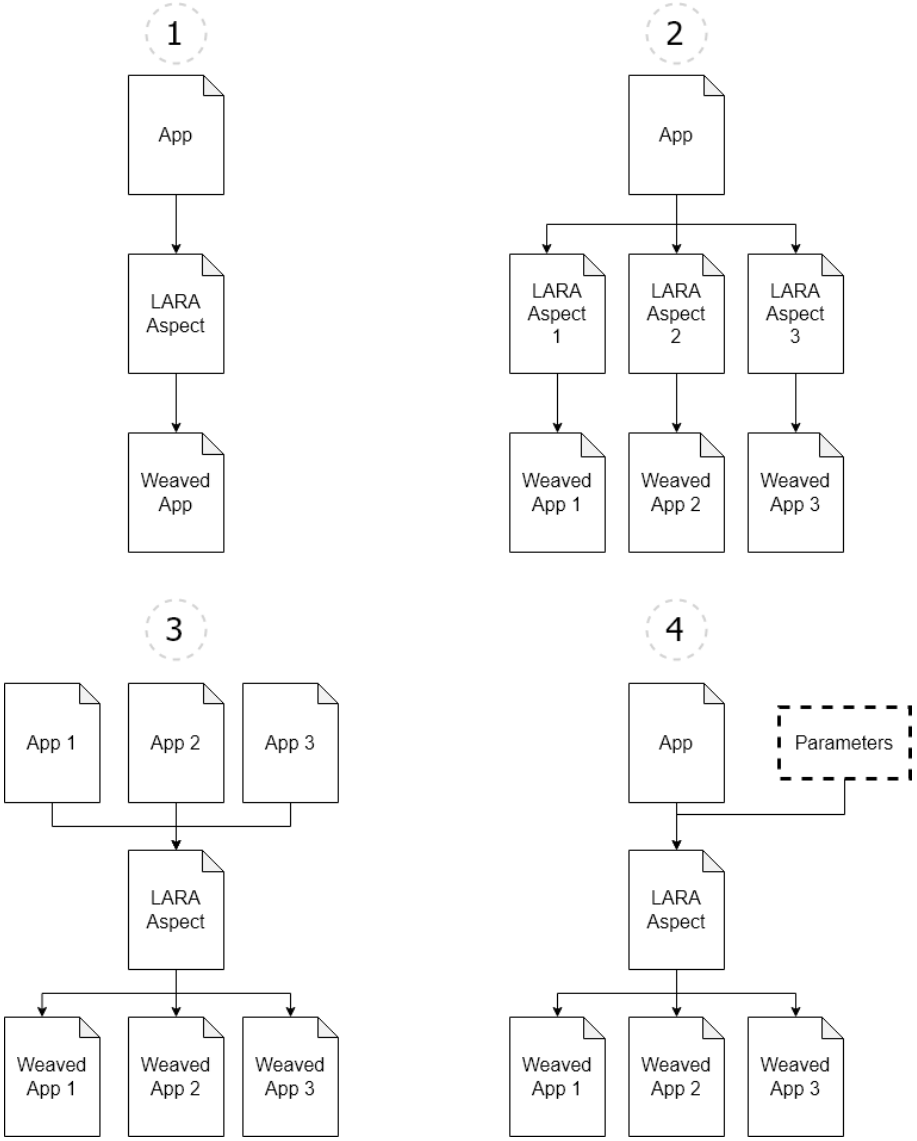

Figure 2.1: Different LARA use cases: A) one aspect applied to one *app* produces one design; B) multiple aspects applied to one *app* produce multiple *apps*; C) one aspect applied to multiple *apps* produces multiple *apps*; D) one aspect, configured with user parameters and applied to one *app*, produces multiple *apps*

2.3 Summary

This chapter presented an overview of modern HLS tools, followed by the description of several recent methodologies of code restructuring for FPGAs.

The overview of HLS tools summarizes existing tools, as well as the patterns, similarities and differences revealed by categorizing the tools based on their properties. Two specific HLS tools were then presented individually. Vivado HLS, a commercial tool, has the advantage of having a very mature development environment, and enjoys of a vast number of possible optimizations. LegUp, an academic open-source tool, has its strengths in adopting higher level constructs of the

C language, as well as being able to seamlessly create hybrid CPU and FPGA applications.

The first article [25], focused on design space exploration, approaches the problem by defining and validating pruning guidelines to choose between designs, as well as an optimizer based on the predicted hardware resources of each remaining design in order to choose a single one. The second article [26] proposes a set of optimizations and the order by which they should be applied, and proves that certain transformations may only have a positive effect on the code after a subsequent transformation is applied. The third article [16] presented describes the construction of a data flow graph from program tracing execution, and then a series of transformations performed over that graph, with the restructured code then being generated from that transformed graph. The fourth article [32] details the conversion of the code onto a data flow template, where parts of the code are converted onto modules that execute in parallel and communicate with each other through queues. The fifth article [33] describes a DSL-based approach to the problem of targeting different HLS tools without having to change the code restructuring directives by proposing a framework, AnyHLS, that allows for code to be written using a platform-independent library. The sixth article [38] centers on a way to balance the trade-off between latency, resource usage and accuracy by finding a series of designs on the Pareto frontier of the three parameters, after generating said designs using a stripped-down representation of source code and applying a series of transformations that simplify arithmetic expressions and memory accesses. Finally, the seventh approach [41] describes the usage of aspect-oriented programming to specify code restructuring strategies on a separate file, which can then be combined together with the source code in order to produce the optimized design. The specific case of the aspect-oriented programming language LARA was detailed, as well as the different use cases enabled by its usage.

Chapter 3

The Code Analysis Framework

In order to analyze source code and decide which optimization strategies can and should be applied, an appropriate framework based on a graph representation of that code needs to be built. This section starts by describing the Clava source-to-source compiler, which is used as the basis to implement the proposed framework. Then, a set of code standardization procedures that are applied to the source code in order to make it more suitable for analysis are described, as well as a functionality that automates the instrumentation step of the trace-based approach proposed by Ferreira and Cardoso [16], which was previously described in Section 2.2.3. Finally, the graph-based code analysis framework is presented in detail, including the control and data flow graph used, the simplification operations that are performed in order to facilitate its analysis and the kind of metrics that can be extracted from those graphs.

3.1 The Clava Source-to-Source Compiler

Clava [15] is a C/C++ to C/C++ compiler developed in the SPeCS group [43]. It is written in Java, and it uses Clang [44] as a frontend. The aspect-oriented language LARA [41] is integrated within the compiler, and both GUI and CLI tools are provided to the developer. As explained in Section 2.2.7, LARA allows for the specification of code transformations, called aspects, by specifying them on a separate file and then applying them to the code through a process called weaving. LARA can also perform extensive code analysis without modifying the source. LARA offers a language specification and application programming interfaces (APIs) that allow for code transformations to be applied at the abstract syntax tree (AST) level, and it allows for new libraries and APIs to be created for it [45].

As an example, let us consider the source code in Listing 3.2, which contains a program with multiple function calls, to which the Instrumentation LARA aspect in Listing 3.1 is applied. This aspect selects all function calls and then, for each one, inserts a print statement just before that

call, in order to log every time a function is called. The code in Listing 3.3 shows the result of the weaving process.

As previously mentioned, Clava uses Clang to parse C/C++ code, and bases its own AST on that of Clang's. This AST is, like most of the Clava codebase, implemented in Java. LARA, however, uses an abstraction layer over Clava's AST based on Join Points. These Join Points also form a tree structure similar to Clava's AST, but allow for higher-level concepts to be expressed, such as the notion of program or file. Both the AST and the Join Points models are used to implement the framework, depending on the level of abstraction necessary. Clava is developed on an open-source GitHub repository [46], and is organized in modules. The modules directly related to the proposed framework are the following:

- ClavaAst - this module holds all classes related to Clava's AST. An analysis submodule is developed in order to allow for a graph representation of source code suitable for HLS-related analysis is implemented here, as well as all the code standardization procedures;
- ClavaHls - a new module for HLS-related analysis and restructuring. It works over the graph obtained from ClavaAst;
- ClavaLaraApi - this module creates APIs that can be accessed by LARA, and it is primarily written in that language. The proposed framework can be accessed by an user through an API called *clava.hls*, which is implemented in this module. A code transformation based on mathematical functions and the possibility to instrument code in order to output a tracing-based data flow graph are also implemented here.

```
1  aspectdef Instrumentation
2    select
3        function.call //Select all functions calls
4    end
5    apply   //Insert print on each selected call
6          $msg = "printf(\"Call to \\\"" + $call.name +
7                  "\\\" in function " + $function.signature +
8                  " at line " + $call.line + "\\n\")";
9
10         $call.insertBefore($msg);
11   end
12 end
```

Listing 3.1: LARA aspect that inserts a print statement before every function call

```
1  double bar() {
2      return 1.0;
3  }
4
5  double fizz() {
6      return 1 + bar();
7  }
8
9  double foo() {
10     int x = bar();
11     for (int i = 0; i < 10; i++)
12         x += fizz();
13     return x;
14 }
15
16 int main() {
17     foo();
18 }
```

Listing 3.2: Example of a C source code that is processed by Clava

```
1  double bar() {
2     return 1.0;
3  }
4
5  double fizz() {
6     printf("Call to \"bar\" in function fizz() at line 6\n");
7     return 1 + bar();
8  }
9
10 double foo() {
11    printf("Call to \"bar\" in function foo() at line 10\n");
12    int x = bar();
13    for(int i = 0; i < 10; i++) {
14       printf("Call to \"fizz\" in function foo() at line 12\n");
15       x += fizz();
16    }
17    return x;
18 }
19
20 int main() {
21    printf("Call to \"foo\" in function main() at line 17\n");
22    foo();
23 }
```

Listing 3.3: Source code after the weaving is performed

3.2 Source Code Preprocessing

This section describes two code preprocessing procedures, which are applied on Clava at the AST level as the first step of the framework. These procedures attempt to simplify some patterns that would complicate code analysis on later stages.

3.2.1 Constant Folding

Constant folding [47] is a code transformation already implemented in many compilers, but it is useful to apply it on an earlier stage here in order to simplify analysis. This optimization attempts, through static code analysis, to detect arithmetic expressions whose all elements are literals (i.e., constant numbers), and replace each found expression by a single literal obtained by evaluating that expression. This is useful when it comes to estimating loop iterations and calculating the index used to access an array. While this transformation could be performed manually for simple cases, its automation becomes a necessity when the constants come from macros, since those can be easily changed between compilations and can be used throughout the entire code, as well as for large codebases, which can offer multiple opportunities for this transformation to be applied. Figure 3.4 shows some examples of constant folding, with some literals that come from macros.

Constant folding is applied in multiple passes over Clava's AST. The C preprocessor has already resolved the macros into their actual expressions on a previous stage, and thus they require no preprocessing. Firstly, the constant folding algorithm searches for all *BinaryOperator* nodes, which is a node type containing an arithmetic operation and with two descendants representing the left and right-hand sides of the operation. For each node, the two descendants are evaluated to check whether they are Literal nodes. A Literal node can be either a *IntegerLiteral* or *FloatingLiteral*, which represent, respectively, integer and real constants with configurable size and precision (e.g., a *FloatingLiteral* can be used to represent both 32-bit and 64-bit floating point numbers). The operator is extracted from the *BinaryOperator* node, and the constants are extracted from each Literal node. Then, and based on the operator, the arithmetic operation between both constants is performed, and the result is stored in a new Literal node. The type of Literal node, as well as the precision and size, are properly configured based on the two constants. Finally, this new Literal node is added to a transformation queue. When all nodes are processed, each Literal node in the transformation queue replaces the *BinaryOperator* node from which it was built, and the constant folding for that instance is done.

However, as previously mentioned, this must be done in several passes, since the constant folding of some expressions may lead to new constant folding opportunities. Given the expression $(1+2)*4$, on a first pass, only the sub-expression $1+2$ would be identified as a constant folding opportunity, since the left-hand side of the $*$ operator is another *BinaryOperator* node corresponding to $+$. Upon transforming $1+2$ into 3, however, the expression would be turned into $3*4$. Now the $*$ operator has both descendants as constants, and a new constant folding opportunity can be identified. Therefore, the constant folding algorithm is applied repeatedly until no new constant folding opportunities can be identified. This example is presented in Figure 3.1.

```
1  #define N 39
2  #define M 20
3
4  //Without constant folding
5  x  = N * M + 1;
6  y[N + 1] = (2 + 4) / 3;
7
8  //With constant folding
9  x = 781;
10 y[40] = 2;
```

Listing 3.4: Examples of constant folding

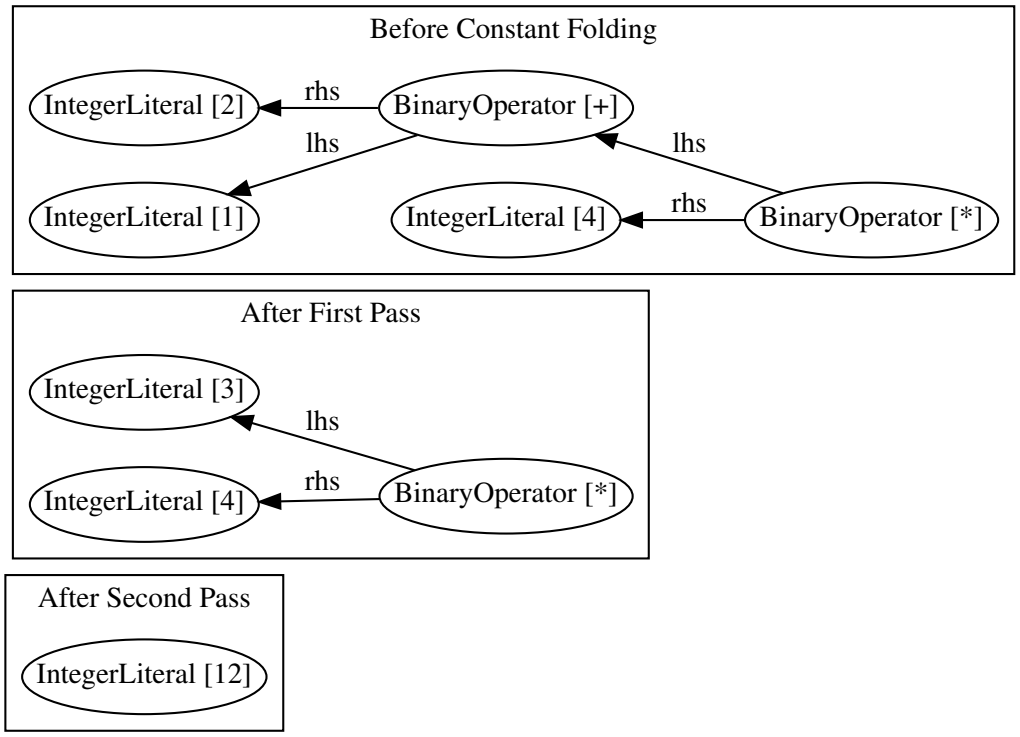

Figure 3.1: Application of constant folding to the expression $(1+2)*3$ through multiple passes over Clava's AST

3.2.2 Unwrapping Statements With Multiple Delarations

Some codebases declare all variables of the same type on a single line, separating them by commas, even when they include some initialization. While this is not a problem when it comes to building a symbol table or find whether a variable exists, it is more useful to the graph construction algorithms if there is only one declaration (with possible initialization) per statement. This conversion is exemplified in Listing 3.5. In Clava's AST, multiple declarations on a single statement

are represented by a *DeclStmt* node, which represents a statement in which one or more variable declarations are performed, and whose direct children represent each of the variables being declared. These children are of the type *VarDecl*, and, if they have any form of initialization, then that *VarDecl* has further children to represent that initialization. The algorithm implemented to unwrap these declarations starts by finding all *DeclStmt* nodes with more than one child. Then, for each *VarDecl* child of that *DeclStmt* node, a new *DeclStmt* is created in order to enclose each child, so that each declaration has its own statement. All new *DeclStmt* nodes are then inserted in the same position as the original *DeclStmt*, and the original one is deleted from the AST.

```
1 //Multiple declarations in a statement
2 int a, b, c = 1, d[10] = {0};
3
4 //One declaration per statement
5 int a;
6 int b;
7 int c = 1;
8 int d[10] = {0};
```

Listing 3.5: Example of unwrapping a statement with multiple declarations

3.3 Tracing-based Data Flow Graph Through Instrumentation

This approach intends to automate the building process of the tracing-based data flow graph originally proposed by Ferreira and Cardoso [16]. This graph represents the data flow of a function by exposing how the data is modified at the function level, during the execution of that function. This was originally done through manual instrumentation of the source code. Print statements have to be inserted before each statement in order to output the graph information when executed on a CPU with no optimizations. This is a laborious and error-prone process, even for small kernels, and automation is necessary in order to make this graph useful for other purposes.

The output produced by instrumentation is in DOT format [48], and it follows the following specification:

- Every time a store operation is performed on a variable, a new node for that variable is created;
- Each variable node holds information about the variable in the form of attributes, such as the type of the variable and whether it is local or an interface (i.e., a function parameter or a global variable);
- Arithmetic operations are represented by a node containing the operator, with two inward edges coming from the two operators. If the operation result serves as the input of another operation, the node is connected to a temporary node;

- For each operation, the edges of the operands hold an attribute indicating whether it is the left or the right operand;
- Every time a constant is used, a new node for the constant is created, even if it had already been used before;
- All accesses to an array position are performed with the index already resolved in runtime (i.e., with a single scalar);
- Arrays of arbitrary size and dimension are supported, as long as those values are known at compile time;
- Ternary operators are represented by a multiplexer node, with three inward edges: one from an expression that evaluates to a boolean, one from the variable to choose if it is true, and another from the variable to choose if it is false. An outward edge connects to the variable on the left-hand side of the assignment;
- Nodes have both an identifier and a label. The identifier is unique for all nodes, but the label is not. For variable nodes, the label is the same for all nodes that refer to the same variable (e.g., the nodes with identifiers "sum_1" and "sum_2" would both have the label "sum"). For operation nodes, the label is the arithmetic operator, and for constant nodes, the label is the constant itself.

An example of a graph is provided in Figure 3.2. It was obtained by rendering the DOT file in Listing 3.8, which was obtained by instrumenting and executing the kernel *row_sum*, whose implementation is on Listing 3.6. A small excerpt of the instrumented *row_kernel* code is provided on Listing 3.7, together with some added comments that explain what each instrumentation print generates. As it is possible to see, the graph is already quite sizeable, even for a small example like this. For more realistic examples with bigger functions and bigger inputs, the resulting graph can be enormous, which leads to scalability issues for the code restructuring tool proposed by the authors.

```
#define N 50

void row_sum(int a[N][N], int b[N]) {
    for (int i = 0; i < N; i++) {
        int sum = 0;
        for (int j = 0; j < N; j++) {
            sum += a[i][j];
        }
        b[i] = sum;
    }
}
```

Listing 3.6: Implementation of a kernel that, for each row of a matrix, gets its sum and stores it on an array

```
//Create a node for the "+" operation
n_op++;
fprintf(log_file_0, "\"op%d\" [label=\"+\", att1=op];\n", n_op);

//Connect a preexisting node accessing "a[i][j]" to the op node as the rhs operand
n_ne++;
fprintf(log_file_0, "\"a[%d][%d]_%d_l\" -> \"op%d\" [label=\"%d\", ord=\"%d\", pos=\"r\"];\n", i, j, n_a[i][j], n_op, n_ne, n_ne);

//Connect a preexisting node accessing "sum" to the op node as the lhs operand
n_ne++;
fprintf(log_file_0, "\"sum_%d\" -> \"op%d\" [label=\"%d\", ord=\"%d\", pos=\"l\"];\n", n_sum, n_op, n_ne, n_ne);

//Create a node representing a store of variable "sum"
n_sum++;
fprintf(log_file_0, "\"sum_%d\" [label=\"sum\", att1=var, att2=loc, att3=int];\n", n_sum);

//Connect the operation node to the store node
n_ne++;
fprintf(log_file_0, "\"op%d\" -> \"sum_%d\" [label=\"%d\", ord=\"%d\"];\n", n_op, n_sum, n_ne, n_ne);

sum += a[i][j]; //The actual statement being instrumented
```

Listing 3.7: Excerpt of the instrumented version of the row_sum kernel, showing the instructions inserted to instrument the statement sum += a[i][j]

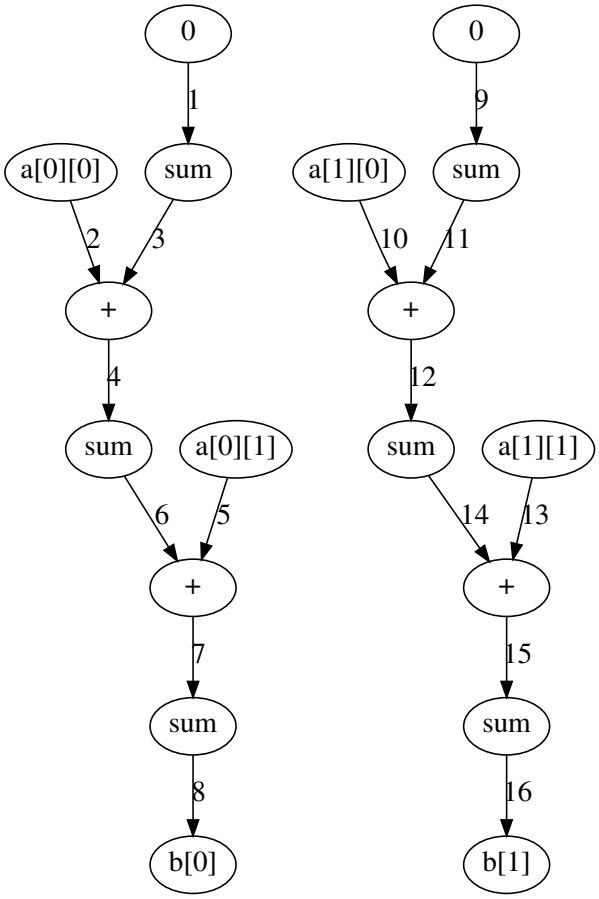

Figure 3.2: Data flow graph obtained by instrumenting and executing the *row_sum* kernel in Listing 3.6 with an input size of N=2

```
Digraph G {
    "a[0][0]_1_l" [label="a[0][0]", att1=var, att2=inte, att3=int];
    "a[0][1]_1_l" [label="a[0][1]", att1=var, att2=inte, att3=int];
    "a[1][0]_1_l" [label="a[1][0]", att1=var, att2=inte, att3=int];
    "a[1][1]_1_l" [label="a[1][1]", att1=var, att2=inte, att3=int];
    "b[0]_1_l" [label="b[0]", att1=var, att2=inte, att3=int];
    "b[1]_1_l" [label="b[1]", att1=var, att2=inte, att3=int];
    "const1" [label="0", att1=const];
    "sum_1" [label="sum", att1=var, att2=loc, att3=int];
    "const1" -> "sum_1" [label="1", ord="1"];
    "op1" [label="+", att1=op];
    ...
    "a[1][1]_1_l" -> "op4" [label="13", ord="13", pos="r"];
    "sum_5" -> "op4" [label="14", ord="14", pos="l"];
    "sum_6" [label="sum", att1=var, att2=loc, att3=int];
    "op4" -> "sum_6" [label="15", ord="15"];
    "b[1]_2_l" [label="b[1]", att1=var, att2=inte, att3=int];
    "sum_6" -> "b[1]_2_l" [label="16", ord="16"];
}
```

Listing 3.8: Abridged version of the DOT file produced by the *row_sum* kernel after being executed with instrumentation and an input size of N=2

Table 3.1: Auxiliary variables created by the automatic instrumentation, where %d refers to the auxiliary variable

Type of node/edge	Auxiliary variable	Node identifier	Label example
Constant	n_const	const%d	0
Temporary	n_temp	temp%d	temp12
Operation	n_op	op%d	+
Multiplexer	n_mux	mux%d	mux7
Variable "foo"	n_foo	foo_%d	foo
Array "bar" of size N	n_bar[N]	bar[index]_%d_l	bar[10]
Edge	n_ne	–	9

The automation tool is implemented in Clava using a LARA aspect. This aspect receives as input the name of the function to instrument, and weaves the provided source code with the print statements, the code to create and close the output DOT file and all the auxiliary variables required in order to keep track of node and edge identifiers. These auxiliary variables are summarized in Table 3.1. While their names and purposes are kept true to the original manual instrumentation for the most part, the node label of a temporary variable is simplified, as it originally included both the line number and the iteration number. While this would not be hard to include, it can lead to the same label being used if there are more than one temporary variable on the same line, and thus it is necessary to change that in order to avoid this conflict. The multiplexer node and the ternary handling are also a posterior addition to the work originally proposed by the authors.

Automation, however, is not without its hurdles. While manual instrumentation allows for some fine tuning when it comes to tracking dependencies on a case-by-case basis, this process becomes a problem when performed automatically. One of the problems comes from the possible reference to a node of a variable that did not yet have a store, and thus no created nodes. This is most commonly found when loading a variable that was passed as a function argument, since most of those already come initialized, and as such there is no preexisting store node to reference. The second problem arises from managing the auxiliary variables on complex arithmetic expressions (i.e., expressions with more than one operand). On those operations, more than one temporary and constant nodes may be created, and the auxiliary variables are incremented accordingly. However, if that happens, it is necessary to introduce offsets when creating the instrumentation prints. Given the example of (a + b) * (c + d), the expression (a + b) will result in a temporary node, as well as the expression (c + d). These two temporary nodes will then serve as input to the multiplication operation, and it is necessary to identify that the temporary node of (c + d) will have its identifier given by the "n_temp" variable, the identifier of (a + b) must be given by "n_temp - 1". This dependency tracking, while simple in this example, can become unwieldy in larger expressions. The proposed automatic instrumentation tool handles both these problems. The tool is divided in four stages, which are therein described:

1. The first step concerns itself with processing the function, and the source code file in general, on a global level. First, from the function declaration the scope and parameters of the

function are extracted. The parameters are then inserted into a symbol table for interface variables, which holds information about the type of each interface variable and, in the case of an array, its dimensions. Then, any global variables that exist in the file are also added to the interfaces symbol table. Finally, the function body is swept through in order to gather all variable declarations, which consist of the local variables of the function. These are added to a symbol table dedicated to local variables;

2. After initializing the symbol tables, the creation of the auxiliary variables is done by going through the symbol table and declaring an auxiliary variable for each entry. Auxiliary variables for the other kind of nodes, such as operations and constants, are also created;

3. In order to circumvent the problem of referencing a node that does not exist (i.e., variables whose first operation is a load), a node is created for every interface variable. If the interface variable is an array, a node is created for each entry in the array. This is done by inserting a loop (or a nest of loops) whose body inserts a print that generates a store node. While this solves the initial problem, there is the instance in which a variable is never loaded (e.g., an array that is passed as an argument and whose purpose is store the result). In this case, the node will never be referenced, since the only operation that is performed on that variable is a store further down the function. These isolated nodes can be pruned once the DOT file is generated by using the *gvpr* tool, which is part of the GraphViz toolkit [49]. The code restructuring framework that uses these graphs also prunes all isolated nodes, so it isn't necessary to run that tool for that particular case;

4. Finally, the instrumentation for the nodes and edges of the function proper are generated. This is done by performing a depth-first search through the function, and by handling each statement it finds independently of all others. All other non-statement Join Point nodes, such as loop headers and comments, are ignored. In Clava, most statements without conditionals have only one child, which is the actual expression from which a data flow can be extracted. These can be variable declarations (possibly with initialization), an unary operation and a binary operation. These can be further divided into array accesses, variable references, literal expressions (e.g., constants) and the operation itself, if the parent expression is a unary or binary operation. The instrumentation prints are inserted individually for each of these instances, and when an operation has complex expressions or other operations as one or both of its operands, the instrumentation is inserted recursively, on a bottom-up fashion and by using, once again, a depth-first exploration of the Join Points. The usage of more than one temporary node is propagated throughout the building process in order to add the proper offsets to them, and thus overcome the previously mentioned limitation. The instrumentation itself is inserted always and immediately before the statement to which it refers to.

This tool is validated with multiple input functions, and some limitations can be found. Some of the limitations found are the impossibility of preserving macros in Clava. This limitation goes

back to it using Clang as a frontend for parsing the code and building an AST, given that Clang runs the C preprocessor to resolve the macros on an earlier step. This is a limitation because it may be necessary to change the size of the inputs of an instrumented function, and while the original function often has macros for such purposes, their loss results in some manual labor in order to either recreate them or substitute all the hardcoded sizes for new ones. The other limitations are the same as the original manual approach when it comes to only accepting arrays of predetermined size, not preserving information about loops and not supporting if-statements, with the exception of the ternary operator, which, as previously mentioned, was a posterior addition to the original proposal. Other constructs of the C language, like type casts, structs and switch statements are also not contemplated. Finally, the limitations imposed by having an FPGA as the target, such as not allowing dynamic memory or recursion, also apply.

3.4 Data Flow Graph Through Static Code Analysis

This section describes the proposed data flow graph (DFG) that will serve as a framework to decide which code restructuring strategies should be applied. Firstly, an overview of the existing control flow graph (CFG) in Clava is provided, followed by the modifications done to it in order to make it suitable to serve as a basis for the DFG construction. Then, a detailed description of the proposed DFG is presented, as well as some pruning and simplification procedures that are applied to it. Finally, an overview is given of the metrics that are possible to extract from the graph.

3.4.1 Control Flow Graph

A control flow graph (CFG) is a directed graph comprised of basic blocks, plus the connections between them. A basic block is a sequence of instructions with one entry point at the beginning and one exit point at the end, while the connections between these basic blocks comprise the control flow paths [50]. Clava already has a prototype that finds basic blocks and the control paths between them, but it lacks an explicit graph structure in terms of implementation. That is fixed with this proposal, as a proper graph representation is implemented over the existing prototype. Therefore, it is now possible to build a CFG that is suitable for analysis. Each basic block in Clava is comprised of a list of statements, and it makes up a node of the CFG. They can be categorized as follows:

- Normal - a basic block with a series of statements. It is the default type;
- Loop - a basic block with a loop header (whether it is a for-loop, a while-loop or a do-while-loop). It can have two kinds of outward edges: one pointing to the first basic block of the loop body, and another pointing to the first basic block that comes after the loop;
- Conditional - a basic block similar to a normal one, but whose last statement holds a statement that implies branching, such as an if-statement. It can have two kinds of outward

edges: one connecting to the first basic block of the code region that should be executed when the expression evaluates to true, and another for when it evaluates to false;

- Exit - a basic block with no outward edges, which represents a termination of the flow. It is otherwise similar to a normal node. In the particular case of a function being comprised of a single basic block, that block has to necessarily be an exit block.

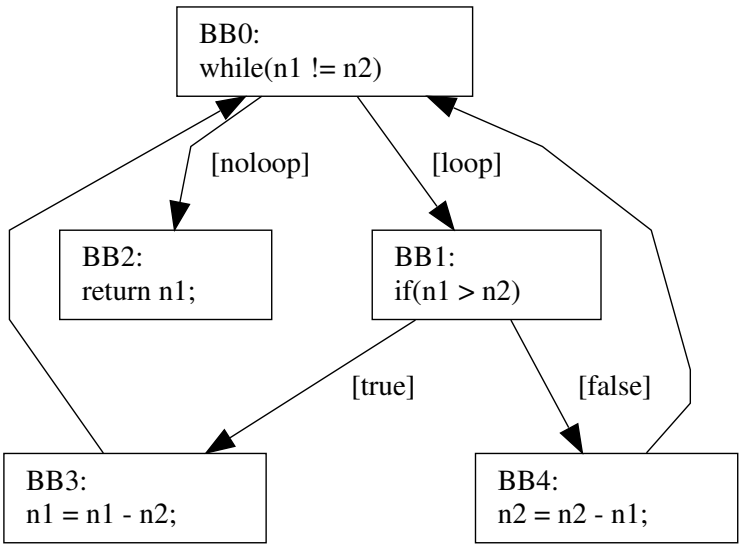

Figure 3.3: Control Flow Graph of the Euclidean algorithm in Listing 3.9

```
1  int gcd(int n1, int n2) {
2      while (n1 != n2) {
3          if (n1 > n2) {
4              n1 = n1 - n2;
5          }
6          else {
7              n2 = n2 - n1;
8          }
9      }
10     return n1;
11 }
```

Listing 3.9: Euclidean algorithm for the greatest common divider between two numbers

Similarly, the control paths of the CFG, which are implemented as the edges of the CFG, can also be categorized as follows:

- Unconditional - a control path between a basic block and another, that is always executed regardless of any condition imposed by the statements inside the first basic block. It is used to connect a normal block to a loop block;

- True - used to connect a conditional block to another block if the statement inside the conditional block evaluates to true;
- False - similar to the true path, but for when the condition evaluates to false;
- Loop - connects a loop block to the first basic block of the loop body;
- No Loop - connects a loop block to the first basic block that comes after the loop.

The CFG can also be formally defined as follows:

CFG definition: the CFG is defined by a set G composed of a set of vertices V and edges E. V is a set composed of four sets. Each of these sets contains vertices of one specific type. BBN contains vertices of representing basic blocks of the "Normal" type, while BBL represents basic blocks of the "Loop" type, BBC basic blocks of the "Conditional" type and BBE basic blocks of the "Exit" type.
E is composed of five sets representing directed edges. U represents edges of the "Unconditional" type, while T represents edges of the "True" type, F edges of the "False" type, L edges of the "Loop" type and finally N represents edges of the "No Loop" type.

$$G = \{V, E\}$$

$$V = \{BBN, BBL, BBC, BBE\}$$
$$BBN = \{bbn_1, ..., bbn_n\}$$
$$BBL = \{bbl_1, ..., bbl_n\}$$
$$BBC = \{bbc_1, ..., bbc_n\}$$
$$BBE = \{bbe_1, ..., bbe_n\}$$
$$\text{E} = \{\text{U, T, F, L, N}\}$$
$$U = \{(u,v) : u \in BBN \wedge v \in BBC\}$$
$$T = \{(u,v) : u \in BBC \wedge v \in V\}$$
$$F = \{(u,v) : u \in BBC \wedge v \in V\}$$
$$L = \{(u,v) : u \in BBL \wedge v \in V\}$$
$$N = \{(u,v) : u \in BBL \wedge v \in V\}$$

To finalize, there are some statements that are not necessary to keep in the CFG, such as comments and directives (other than a few select ones). These are pruned from each basic block after the CFG is generated, but are still preserved in the AST. Figure 3.3 shows the CFG of the Euclidean algorithm to calculate the greatest common divider of two numbers, and whose implementation can be found in Listing 3.9. This example provides a simple visualization of all node and edge types of the CFG, since it has both conditional jumps and loops.

3.4.2 Data Flow Graph

One of the major issues with the previous trace-based DFG is that its size can get unwieldy with even moderate input sizes, which can cause an out-of-memory error on the tool that examines it. Other major issues are the lack of support for conditional statements and the fact that it does not preserve any information about loops. This section proposes an alternative DFG, obtained using the CFG, that is capable of representing the data flow of a function without those limitations, and which can be used to do a different kind of code analysis and restructuring than the one done over the tracing-based graph.

This section gives an overview of the elements of the proposed DFG. One example of a simple DFG is provided in Figure 3.5, which is obtained from the matrix multiplication kernel present on Listing 3.11. This kernel is extracted from the UTDSP [30] benchmark suite, and it multiplies the matrices *a_matrix* and *b_matrix*, placing the result on *c_matrix*. All matrices have a dimension of 10×10. A more complex example is presented in Figure 3.4, with the corresponding source code in Listing 3.10. This example shows the implementation of a Discrete Cosine Transform (DCT), which is performed over 8×8 matrices. The DFG can also be formally defined:

```
1  #define N 8
2
3  const int CosBlock[N][N] = {88, 122, /*abridged*/ 47, -24};
4
5  void dct(int InIm[N][N], int TempBlock[N][N],
6           int CosTrans[N][N], int OutIm[N][N]) {
7      int aux;
8
9      for (int i = 0; i < N; i++)
10         for (int j = 0; j < N; j++) {
11             aux = 0;
12             for (int k = 0; k < N; k++)
13                 aux += InIm[i][k] * CosTrans[k][j];
14             TempBlock[i][j] = aux;
15         }
16
17     for (int i = 0; i < N; i++)
18         for (int j = 0; j < N; j++) {
19             aux = 0;
20             for (int k = 0; k < N; k++)
21                 aux += TempBlock[k][j] * CosBlock[i][k];
22             OutIm[i][j] = aux;
23         }
24 }
```

Listing 3.10: Implementation of a Discrete Cosine Transform (DCT) using matrices

DFG definition: the DFG is defined by a set G composed of a set V of vertices and E of edges. V is composed of multiple sets: LD is a set of "Load" nodes, ST is a set of "Store" nodes, OP is a set of "Operation" nodes, L is a set of "Loop" nodes, I is a set of one "Interface" node, C is a set of "Constant" nodes, and T is a set of "Temporary" nodes. LD, ST and OP are composed by subsets representing each subtype. LD is composed of a set LV of "Load Variable" nodes, a set LA of "Load Array" nodes and a set LI of "Load Index" nodes. ST is composed of a set SV of "Store Variable" nodes and a set SA of "Store Array" nodes. Finally, OP is composed of a set OA of "Arithmetic" nodes, a set of OC of "Conditional" nodes, a set OF of "Function call" nodes and a set OM of "Multiplexer" nodes. E is composed of three subsets, representing three types of directed edges. DF represents edges of the "Data flow" type, IDF edges of the "Index Data Flow" type and "RE" edges of the "Repeating" type.

$$G = \{V, E\}$$

$$V = \{LD, ST, OP, L, I, C, T\}$$
$$LD = \{LV, LA, LI\}$$
$$LV = \{lv_1, .., lv_n\}$$
$$LA = \{la_1, ..., la_n\}$$
$$LI = \{li_1, ...mli_n\}$$
$$ST = \{SV, SA\}$$
$$SV = \{sv_1, ..., sv_n\}$$
$$SA = \{sa_1, ..., sa_n\}$$
$$OP = \{OA, OC, OF, OM\}$$
$$OA = \{oa_1, ..., oa_n\}$$
$$OC = \{oc_1, ..., oc_n\}$$
$$OF = \{of_1, ..., of_n\}$$
$$OM = \{om_1, ..., om_n\}$$
$$L = \{l_1, ..., l_n\}$$
$$I = \{i_1\}$$
$$C = \{c_1, ..., c_n\}$$
$$T = \{t_1, ..., t_n\}$$
$$E = \{DF, IDF, RE\}$$
$$DF = \{(u, v) : u \in \{O, L, C, T\} \wedge v \in \{O, S, T\}\}$$
$$IDF = \{(u, v) : u \in \{O, L, C, T\} \wedge v \in LA\}$$
$$RE = \{(u, v) : u \in \{L, I\} \wedge v \in \{L, S, OF\}\}$$

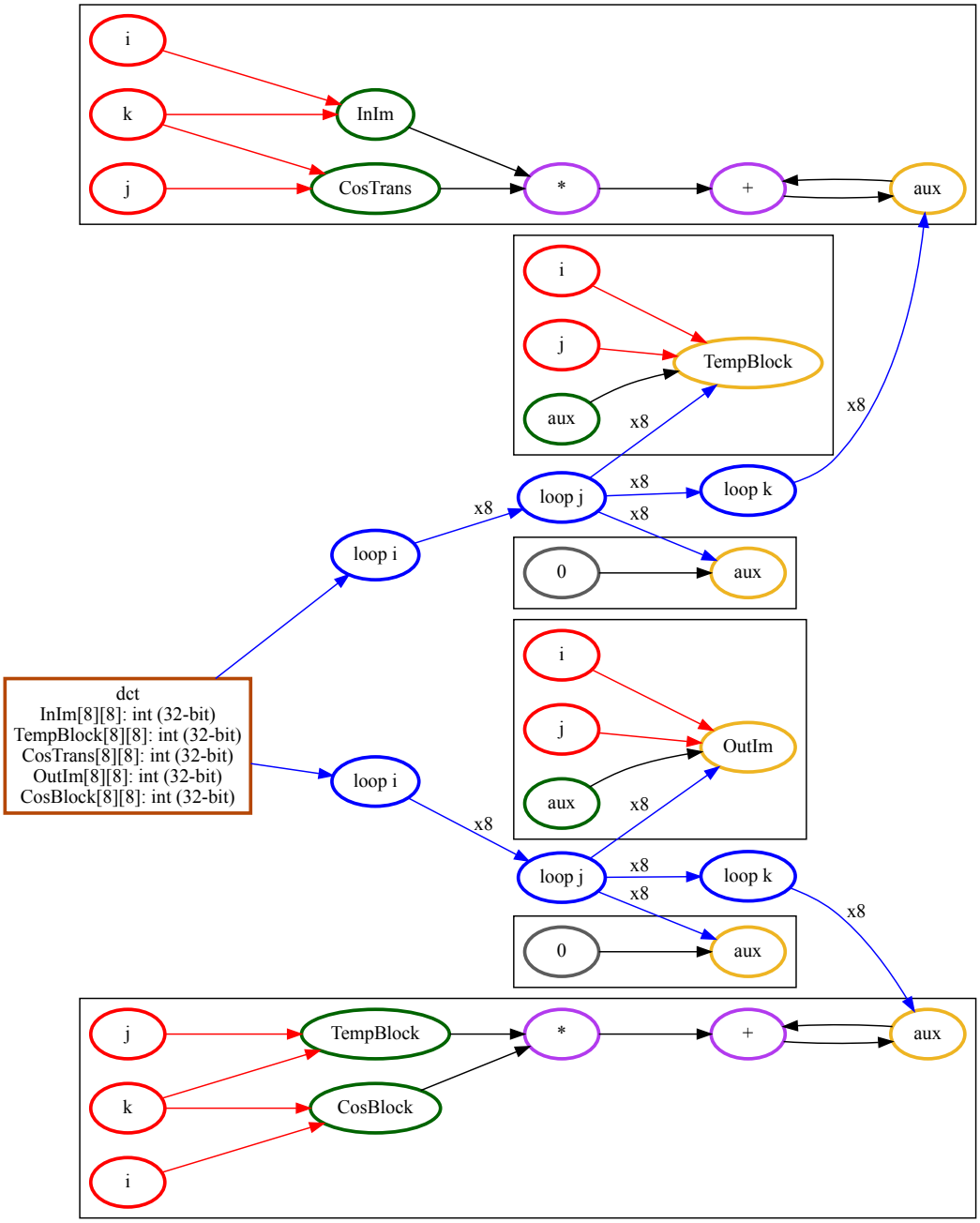

Figure 3.4: Data flow graph of the DCT benchmark. The square node represents the root, blue nodes represent loops, yellow nodes represent store operations, green nodes represent load operations, red nodes represent the load of a loop counter, gray nodes represent constants and purple nodes represent operations

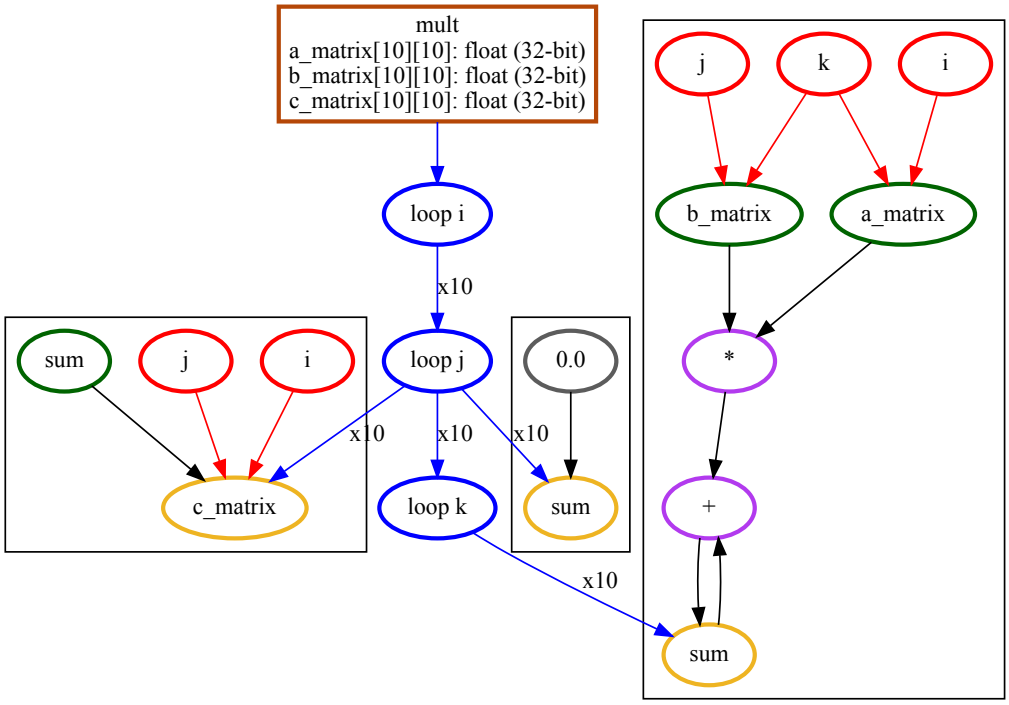

Figure 3.5: Data flow graph of a matrix multiplication kernel. The square node represents the root, blue nodes represent loops, yellow nodes represent store operations, green nodes represent load operations, red nodes represent the load of a loop counter, gray nodes represent constants and purple nodes represent operations

```
1  #define N 10
2
3  void mult(float a_matrix[N][N],  float b_matrix[N][N],
4            float c_matrix[N][N]) {
5      float sum;
6
7      for (int i = 0; i < N; i++) {
8          for (int j = 0; j < N; j++) {
9              sum = 0.0;
10             for (int k = 0; k < N; ++k) {
11                 sum += a_matrix[i][k] * b_matrix[k][j];
12             }
13             c_matrix[i][j] = sum;
14         }
15     }
16 }
```

Listing 3.11: Implementation of a matrix multiplication kernel from the UTDSP benchmark suite

There are three types of edges on the DFG. The first type, "Data Flow", is a typical data flow edge, which is a directed edge that represents the transfer of data between variables/operators.

Table 3.2: Types of DFG edges

Type	Color
Data Flow	Black
Index Data Flow	Red
Repeating	Blue

This is the type of edge used to represent most data flows. The second type, "Index Data Flow", is similar to the Data Flow edge, except that it serves to connect a subgraph that calculates an array index to the node that represents the array access. Finally, the "Repeating" edge type is an edge that indicates how often the data flow subgraph on its destination is repeated. All data flows have one incoming "Repeating" edge, even those that are only executed once. Each "Repeating edge" has a label representing the number of repetitions it represents (which is omitted when it equals 1). All three edge types are listed on Table 3.2, where their respective color codes can be found.

The nodes of the DFG can have different types, represented as different colors on the generated DOT output. These are gathered in Table 3.3. Each type can be further divided into subtypes. Each node can only have one type, and only one subtype within that type. The first node type serves to represent a read, or loading, operation of a variable. It is divided into 3 subtypes: scalar variables are represented by the "Load Variable" subtype, while an array access is represented by a "Load Array" subtype. The exact array position being accessed is given by the index, which can be the result of a previous calculation, and it is associated to the "Load Array" node by an "Index Data Flow" edge. Finally, the scalar variables used as the loop counter are identified by their own type, "Load Index".

There are two subtypes of store nodes: a "Store Variable" node represents the operation of writing a value to a scalar variable, while a "Store Array" node represents the operation of writing a value to an array position. In the same fashion as the "Load Array" node, the index is given by an incoming "Index Data Flow" edge.

Operations consist of four different subtypes. The "Arithmetic" and "Conditional" subtypes represent operations performed using arithmetic and logical operators, respectively. The operands come from incoming edges, and an outgoing edge connects the result of the operation to the node that makes use of it. The "Function call" subtype represents a call to a function. If the function has operands, then the node will have incoming edges holding each operand, and if the function has a return value that is not ignored, then the node will have an outward edge to connect it to the node that uses the result. Finally, the "Multiplexer" node represents a selection of some data flows over others when taking into consideration the result of a condition. It can have three different types of incoming nodes: one from a data flow that consists of a condition, multiple nodes from data flows that are selected when the condition is true, and multiple data flows that are selected when the condition is false. If the "Multiplexer" node was used to implement a ternary assignment operator, then the node has an outward edge to connect it to the variable being assigned.

Finally, there is a group of miscellaneous nodes that are used to provide extra information to the DFG. There is one and only one "Interface" node per DFG. This node serves as the root for

Table 3.3: Types of the DFG nodes

Type	Subtype	Color	Label Example
Load	Variable	Green	sum
	Array	Green	a_matrix
	Index	Red	i
Store	Variable	Yellow	sum
	Array	Yellow	c_matrix
Operation	Arithmetic	Purple	+
	Conditional	Purple	!=
	Function call	Purple	sqrt
	Multiplexer	Purple	mux
Loop		Blue	Loop i
Interface		Brown	mult
Constant		Grey	0
Temporary		Grey	temp1

the graph, and holds information about the function name and its interface variables (i.e., function arguments and referenced global variables), as well as the size of their respective data types. The "Loop" nodes represent a loop, holding information about which variable is the loop index, the loop lower and upper bound and the index increment value. They have outward edges of the type "Repeating", which connect to data flows or other "Loop" nodes, and have a value equaling the number of iterations of the loop they refer to. The "Constant" nodes represent literal constants, which can be both integers or floating-point values. Finally, the "Temporary" nodes serve to store the temporary results of arithmetic operations. By default, these nodes are not generated, since the analysis currently being performed over the DFG does not use them. However, they could be useful for alternative approaches that call for temporary results to be stored on their own local variables.

3.4.3 Building Process

The first node built is the "Interface" node that serves as the root of the DFG. As previously explained, this node holds information about the function name and the function interface, which is comprised of arrays passed as function arguments and all global arrays referenced by the function. Scalar function arguments are not considered. The declarations of these variables are identified by looking at the AST. For each variable, multiple parameters are extracted by looking at the declaration. One of these parameters is whether it is an array or a scalar variable. If it is an array, information about the dimension and the size of each dimension are also stored. Another parameter is the data type of the variable, which can be further distinguished into being an integer (e.g., char, short, int) or floating-point (e.g., float, double) data type. The size of each data type,

specified in bits, is also recorded. The framework supports this analysis for all native C data types, but support for custom type definitions are not contemplated.

```
1  int find_max(int in0, int in1, int in2)
2  {
3      int max;
4      if (in0 > in1)
5      {
6          max = (in0 > in2) ? in0 : in2;
7      }
8      else
9      {
10         max = (in1 > in2) ? in1 : in2;
11     }
12     return max;
13 }
```

Listing 3.12: Implementation of a function that finds the maximum of three numbers, using both if-else and ternary statements

The nodes of the DFG are built from examining the AST nodes, while using the CFG as an intermediate layer between the building algorithm and the AST. In order to find the nodes that directly descend from the "Interface" node, the top-level control flow path of the function (i.e., the sequence of basic blocks that represent operations on the uppermost scope of the function) must be determined. This is done by selecting the first basic block of the CFG, and descending the CFG from there. If a basic block of the "Loop" type is found, the next descendant will be the one pointed at by the edge of the type "No Loop". If a "Conditional" basic block is found, the next top-level block is defined as the first basic block common to the control flows pointed at by the "True" and "False" edges (i.e., when branching execution paths join after being split). As an example, the top-level control flow path of the CFG in Figure 3.3 would be BB0 $\rightarrow$ BB2. From this top flow, the data flow subgraphs for each basic block are generated, and they are all connected to the "Interface" node through a "Repeating" edge with a value of 1, since each of these data flows, being at the top of the function, are only executed once per function.

Each basic block is dealt with differently when it comes to building their data flows. A "Normal" basic block is processed by building the data flow for each of the statements inside that block. A "Loop" block is processed by creating a node of type "Loop", and then recursively applying the same process to the basic blocks of the control flow path pointed at by the "Loop" outward edge. After the data flows from these innermost basic blocks are generated, they are connected to the "Loop" node by a "Repeating" edge with a value equaling the number of iterations of the loop (the estimation of the number of iterations is further defined in section 3.4.4). Finally, a "Conditional" basic block is processed by creating a "Multiplexer" node. The data flow for the conditional expression (which comes from a statement inside the "Conditional" basic block) is generated and connected to the multiplexer. Then, this process is applied recursively to the control flow paths

pointed at by the "True" and "False" edges of the "Conditional" block. Once those data flows are calculated, they are connected to the "Multiplexer" with a standard data flow edge, but with an additional flag to indicate whether they should or not be selected based on the multiplexer condition. Figure 3.6 shows the DFG of the function in Listing 3.12, in which it is possible to observe "Multiplexer" nodes built from both if-else and ternary statements. Note, as well, the labels assigned to the edges incising on each multiplexer, where "s" stands for "selection", "t" stands for "true" and "f" stands for "false".

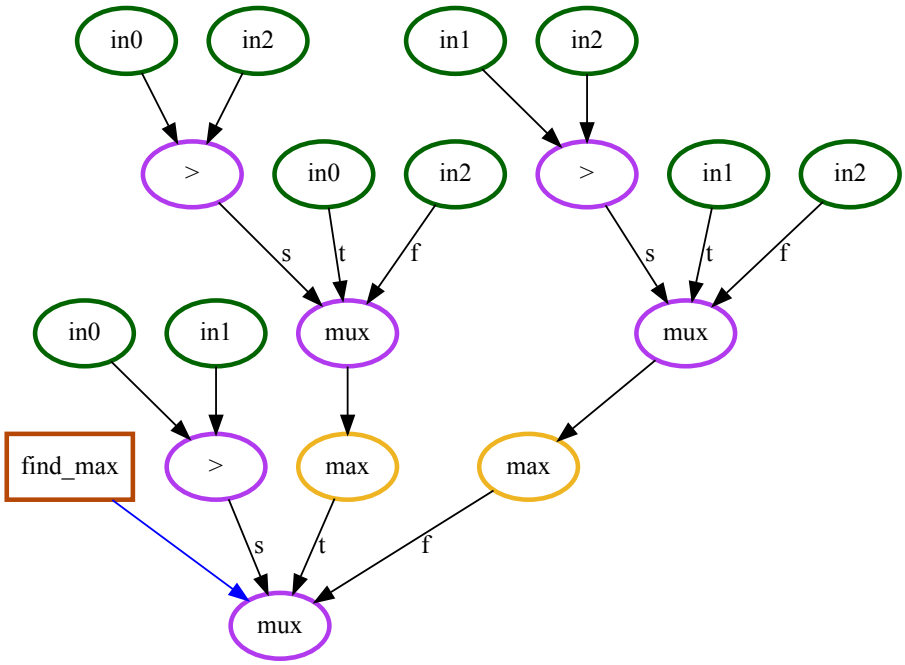

Figure 3.6: Data flow graph of the find_max function. The square node represents the root, yellow nodes represent store operations, green nodes represent load operations, red nodes represent the load of a loop counter, gray nodes represent constants and purple nodes represent operations

As previously mentioned, a data flow subgraph is generated from a statement extracted from a basic block. The statement is represented by an AST node of the *Stmt* type. Each *Stmt* type, on the contemplated cases, have only one child AST node. This AST node goes through a dispatcher that, based on the type, is sent to a handler that builds the DFG that better implements that AST node. Depending on the type of node, and on its children nodes, a recursive approach must be adopted, with the dispatcher being called for each child AST node until the AST of the expression is consumed and the data flow subgraph for that statement is complete. The AST node types being processed by the dispatcher are the following:

- *IntegerLiteral* - represents an integer constant. It leads to the creation of a "Constant" DFG node. It is a terminal node;
- *FloatingLiteral* - it has the same behavior as an *IntegerLiteral*, except that it represents a floating-point constant;

- *DeclRefExpr* - represents a reference to a variable. It leads to the creation of a "Load Variable", "Load Index" or "Load Array" DFG node. It is a terminal node;
- *ArraySubscriptExpr* - represents the array access expression that calculates the array position. It is a node that requires recursive building of the DFG of its children. It is connected by a "Data Flow Index" edge to a "Load Array" or "Store Array" node;
- *BinaryOperator* - represents an arithmetic or logical operation with two operands, plus an optional node in which the result is stored (which leads to the creation of a "Store Variable" or "Store Array" node). It leads to the creation of an "Arithmetic" DFG node. Recursive building of its left and right operands must be performed, and the DFGs resultant from that construction is connected to the created "Arithmetic" node;
- *UnaryOperator* - represents an unary operator being applied to a variable. This usually requires the unary operator to be converted into a binary operation in order to expose the data flow. For instance, a post-increment operation over a variable, such as i++, needs to be converted into i = i + 1, and the sign inversion operation, such as -x, is converted into -1 * x. It requires recursive dispatching for the operand;
- *CallExpr* - represents a function call. A "Function call" DFG node is created, and the function arguments, as well as the variable in which the function result is stored (if present), need to be build recursively and connected to the created "Function call" node;
- *CStyleCastExpr* - represents a casting operation over a variable or expression. This node is ignored, since casts are not contemplated in the data flow graph. The variable or expression being cast is instead processed as if the cast did not exist;
- *ParenExpr* - represents the parenthesis that encapsulate an expression. Similarly to the *CStyleCastExpr*, this node can be safely ignored and the expression it encapsulates is also handled like all others, since the priority implied by the parenthesis is respected in the DFG;
- *ConditionalOperator* - similar to the *UnaryOperator*, but for logical operations instead;
- *VarDecl* - represents the declaration of a variable, which is represented by a "Store Variable" or "Store Array" type. It is only handled if there is initialization, since variables with that are simply declared, with the initialization happening on a later stage, can be ignored. It requires recursive building for the initialization expression.

Finally, it is worth noticing the features of the C language that the proposed DFG does not support. The usual restrictions of FPGAs are still valid: no dynamic memory, recursion and pointers, but on top of those, the usage of structs, switch statements and custom type definitions are also not contemplated. Due to Clava being also capable of handling C++, the DFG can also handle C++ code, as long as it only uses features that also exist in C (e.g., no templates, objects, namespaces and other features specific to C++ are supported).

3.4.4 Loop Iteration Estimation

On Clava, for-loops are represented on the AST by the node type *ForStmt*, which can have four children: three statements of the that comprise the loop header, and a *CompoundStmt* that represents the loop body. In order to determine the number of iterations of the loop, all four children must be examined in order to extract a series of values:

- Loop counter - the loop counter, or index, is the variable that is incremented or decremented on each iteration, and commonly used in the expressions that access an array position. The loop counter is found by looking at the first statement of the loop header, which is usually either a variable declaration (e.g., int i = 0), or a reference to an existing variable with an assignment (e.g., i = 0). In either case, the loop counter is assumed to be the variable on the left-hand side of the expression;

- Initial value - the initial value is the value of the loop counter on the first iteration of the loop. It is assumed to be a constant value on the right-hand side of the first statement of the loop header;

- Final value - the final value is the value of the loop counter after the last iteration of the loop. It can be found by examining the conditional expression of the loop. This condition must have a logical operator with the loop counter on one side and a constant on the other, regardless of the order. The constant value provides the final value of the loop counter, and the kind of logical operator provides information about whether the loop counter increases or decreases throughout the execution of the loop;

- Loop counter increment/decrement - the number of units that the loop counter is incremented or decremented by on each iteration. This is done by looking at the third expression of the loop counter, which can be either a binary or unary operation (represented by a *BinaryOperator* and *UnaryOperator* node on the AST, respectively). The former must represent a binary expression that adds or subtracts a constant to the loop counter (e.g., i += 2), while the latter can be either a pre/post increment/decrement operator (e.g., i++), which means that the loop counter increment/decrement is unitary.

The number of iterations is, then, given by $(final\_value - initial\_value)/increment$ if the loop counter is increased during the loop execution, or $(initial\_value - initial\_value)/decrement$ if it decreases. However, this approach cannot detect the number of iterations of loops when there is one or more factor that is only known at runtime (e.g., the final value may be stored on a variable instead of being a constant). In this case, it is still possible for a developer to manually specify the number of iterations of a loop by using a custom directive, *#pragma MAX_ITER N*, where N is the number of loop iterations. This directive should be applied immediately before the for-loop statement.

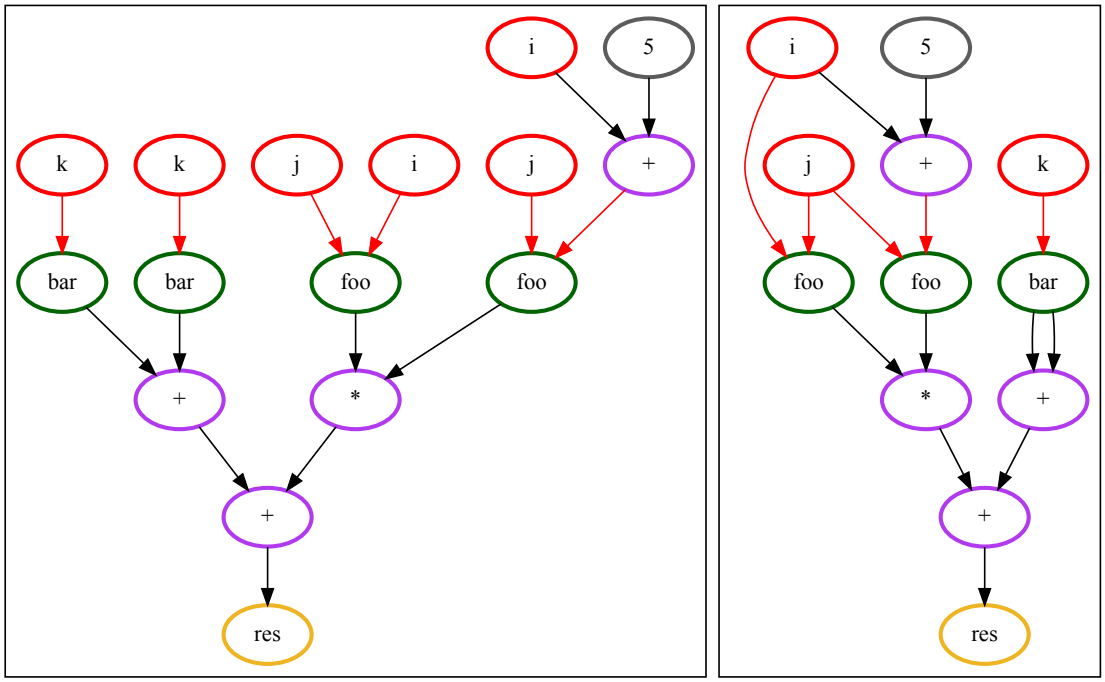

Figure 3.7: Example of a data flow subgraph before and after the simplification process

3.4.5 Simplifying the Data Flow Graph

The data flow graph, after being generated, undergoes a simplification process in order to reduce the number of redundant nodes. The first step in this process tries to identify a set of data flow subgraphs. A subgraph is defined as the set of data flow nodes and edges whose sink is either a "Store" node or a "Function call" node whose return value is ignored. On the matrix multiplication example in Figure 3.5, these subgraphs can be identified by the rectangles encapsulating all nodes, with the exception of the "Interface" and "Load" nodes, which, by definition, are never part of any subgraph. Each node that belongs to a subgraph is tagged with the identifier of the subgraph it belongs to. This is done by first finding all sinks of the previously mentioned types, and assigning a unique sequential identifier to each one. Then, for each sink, the DFG is traversed upwards, from sinks to sources, and all nodes found are tagged with the same identifier as the original sink. Given that there is no dependency tracking at this point, it is not possible for conflicts to arise.

Then, for each data flow subgraph, duplicated nodes are removed. A duplicated node constitutes of a "Load" node that appears more than once. Identifying duplicated nodes is trivial for nodes of the type "Load Variable" and "Load Index", since the nodes can be compared by the variable name alone. However, for nodes of the type "Load Array", there is a need to look at the expressions that calculate the array index in order to estimate whether it is a load of the same array position. This is done by comparing the data flow that makes up each array access (i.e., the data flow connected to the "Load Array" node by a "Index Data Flow" edge). Once again, this detection is limited by what is possible to know at compile-time. Two accesses are considered to be the same if the data flow of the accesses reference the same counter and have the same offset (e.g., i+1

and 1+i are correctly identified as being the same access). This is where constant folding shows its usefulness, since it helps simplifying many array access expressions. More complex expressions, however, are not possible to be detected, either due to high complexity or due to values that are only known at runtime. Once all duplicated nodes are detected, they are merged into a single node. This is done, for each set of duplicated nodes, by picking one random node as the pivot. Then, the inward and outward edges of the remaining nodes are added to the pivot, and those remaining nodes are eliminated. Figure 3.7 shows an example of the simplification process of the data flow subgraph of the expression res = (bar[k] + bar[k]) + (foo[i][j] * foo[i + 5][j]). As it is possible to see, all loads of array indexes merge into a single load per variable, and the access to the "bar" array was merged into a single one. The accesses to the "foo" array, however, were not merged, since the array access patterns are distinct, and therefore represent different array positions.

3.4.6 Extracting Features

Given that there are many different types of DFG nodes, it is possible for the framework to characterize each data flow subgraph using the number of DFG nodes of each type. Table 3.4 shows the features extracted for each data flow subgraph of the svm_predict function used in the article by Tsoutsouras et al. [25] and Ferreira and Cardoso [16]. Its implementation can be found in Listing 3.13. Each subgraph is represented by its identification number, and they can be observed in Figure 3.8, which shows the full DFG for the svm_predict function. The subgraph identification numbers are usually omitted, but they were added to this example for clarity purposes. For each subgraph, the following features are extracted:

- Iterations - the number of times that data flow is executed. It is calculated by recursively finding all nodes of the "Loop" type that are ancestors of the data flow, until the "Interface" node is found. The final number of iterations is given by multiplying the number of iterations provided by the edges of the "Repeating" type that connect the subgraph to the loop nodes (e.g., subgraph 4 has 2 "Loop" ancestors, and its number of iterations is given by multiplying the values of the three "Repeating" edges (i.e., blue edges) connecting those nodes, that is, $18 \times 1248 \times 1 = 22464$);

- LoadsVar - the number of load operations performed over a scalar variable. It is calculated by counting every node of the "Load Variable" and "Load Index" types. However, when the same scalar variable is both loaded and stored on the same subgraph, that variable is only represented by a "Store Variable" node, as the load operation is made implicit by having an outward edge leaving the "Store Variable" node. These situations can be detected, and the value of the LoadVar feature is incremented by one if it happens. This can be observed on the "Store Variable" node of the "norma" variable of subgraph 4;

- LoadsArr - the number of load operations done over distinct array positions. It can be calculated by counting the number of "Load Array" nodes. The simplification process performed

Subgraph ID	Iterations	LoadsVar	LoadsArr	StoresVar	StoresArr	Ops	Calls	Depth
1	1	0	0	1	0	0	0	0
2	22464	2	2	1	0	1	0	1
3	22464	1	0	1	0	1	0	1
4	22464	2	0	1	0	1	0	1
5	1248	3	1	1	0	4	1	5
6	1248	0	0	1	0	0	0	0
7	1	1	0	1	0	1	0	1

Table 3.4: Features extracted from the DFG of svm_predict

on section 3.4.5 already accounts for all "Load Array" nodes within a subgraph to be representing the loading of different array positions, so they can be simply summed up. The same special case of having both a load and a store is also contemplated;

- StoresVar - the number of stores performed over a scalar variable on a single execution of a subgraph. It can be either 1 or 0, since each data flow subgraph has a store at its root, with the exception of subgraphs built from function calls whose return value is ignored;
- StoresArr - the number of stores performed over distinct array positions. It follows the same binary logic as the previous feature;
- Ops - the number of operations on the subgraph. It is done by counting all "Conditional" and "Arithmetic" nodes;
- Calls - the number of function calls on the subgraph, calculated by counting all "Function call" nodes;
- Critical Path - the critical path of a subgraph is defined as being the acyclic sequence of nodes with the most operations. Table 3.5 shows the critical paths calculated for each subgraph of the svm_predict function. The critical path is calculated by traversing the subgraph from its sink (which, by definition, is a "Store" node or a "Function call" node) using breadth-first search. At each level of the search, the number of operations on each possible path is updated, and once the algorithm terminates, the path with the longest value is chosen;
- Depth - the number of operations of the critical path.

Table 3.5: Critical paths of the svm_predict data flow subgraphs

Subgraph ID	Critical Path
1	0 → sum
2	j → test_vector → "-" → diff
3	diff → "*" → diff
4	diff → "+" → norma
5	-1 → "*" → "*" → exp → "*" → "+" → sum
6	0 → norma
7	0 → sum

```
#define GAMMA 8
#define B 0
#define N_FEATURES 18
#define N_SUP_VECT 1248

int svm_predict(float test_vector[N_FEATURES],
                float sup_vectors[N_FEATURES][N_SUP_VECT],
                float sv_coeff[N_SUP_VECT])
{
    float diff;
    float norma;
    int sum = 0;
    for (int i = 0; i < N_SUP_VECT; i++)
    {
        for (int j = 0; j < N_FEATURES; 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;
    return sum;
}
```

Listing 3.13: Implementation of the svm_predict function

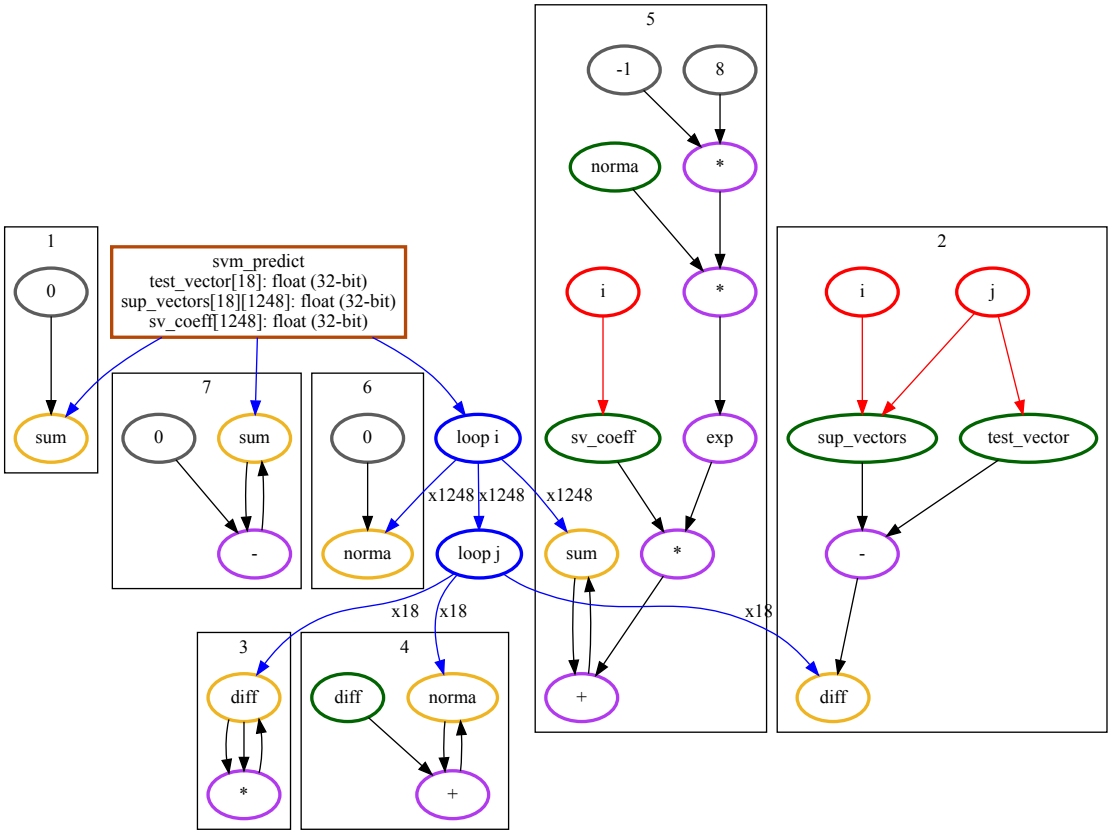

Figure 3.8: Data flow graph of a SVM prediction function. The square node represents the root, blue nodes represent loops, yellow nodes represent store operations, green nodes represent load operations, red nodes represent the load of a loop counter, gray nodes represent constants and purple nodes represent operations. Note, too, that each subgraph is numbered

3.5 Summary

This chapter presented the Clava Source-to-Source C/C++ compiler as the basis on which the proposed framework is implemented, as well as the AOP LARA language. Then, two code preprocessing steps were described: one based on constant folding, which reduces expressions composed of multiple constants into one single constant, and one based on transforming code in order to have, at most, one variable declaration or initialization per statement. Following that, a tool that automates the instrumentation step of the trace-based approach proposed by Ferreira and Cardoso [16] was described in detail. This tool is implemented in LARA, and it inserts instrumentation prints on the targeted function. These prints, when the code is executed, output a DFG that can be then transferred to the code restructuring tool proposed by the authors. Finally, the proposed code restructuring framework itself was explained. A CFG is defined, followed by a DFG that is built from that CFG. This DFG models the data flow of a function by representing load/store operations over variables, arrays and indexes, as well as function calls, arithmetic and conditional operators and information about loop iterations. The building process of this DFG is described, together with the C subset that is currently being supported. Then, the calculation of the number of iterations of a loop is explained. This calculation supports loops whose index either ascends or descends, as well as loops with increments different from 1. If this calculation is not possible at runtime, it is still possible for the user to provide information about the number of iterations through a custom directive. After that, the simplification process of the graph is detailed. This process merges the load operations of variables and array positions, with further analysis done in order to decide whether an access to an array with an index unknown at compile-time is the same as another similar access. Finally, a number of features can also be extracted from the DFG, such as the number of load/store operations on each variable and array, the number of arithmetic operations and function calls, the critical path of a subgraph and the number of times each subgraph is executed.

Chapter 4

Code Optimization Strategies

This chapter describes the code optimizations performed by analyzing the DFG. This process is described in Figure 4.1, where two optimization flows, A and B, can be identified. Flow A represents the main code optimization flow, and is comprised of a set of strategies applied in a predetermined order.

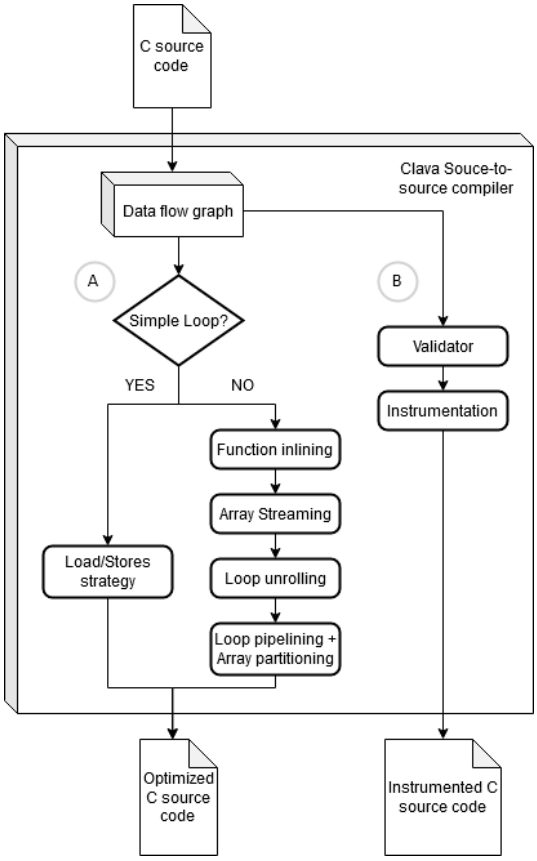

Figure 4.1: Stages of code optimization applied by the framework

These strategies are based on the selection, configuration and insertion of Vivado HLS directives, with heuristics guiding the decision process. The strategies comprise of function inlining, modeling input arrays as streams, loop unrolling, loop pipelining and array partitioning, which are applied over a generic input. However, if the input is a simple loop, then a different strategy based on exposing concurrent load/store operations is applied instead.

Alternatively, it is also possible to check if the code can be also restructured by the trace-based tool proposed by Ferreira and Cardoso [16], and instrumented by the instrumentation tool proposed in Section 3.3. These two steps are exemplified by flow B in Figure 4.1.

Finally, there is a code restructuring strategy based on mathematical functions and data type manipulation, implemented in LARA, that can be applied independently or in conjunction with any of the two options.

4.1 Restructuring Using Vivado HLS Directives

This section details five code restructuring strategies that are applied using Vivado HLS directives [23], together with the heuristics that help decide where and when they should be applied. These strategies comprise of function inlining, loop unrolling, pipelining code regions, optimizing simple loops and modeling input arrays as FIFOs. Finally, a reflection about how the framework could support other unused Vivado HLS directives is provided. All these strategies are implemented in the ClavaHls module of Clava.

4.1.1 Function Inlining

Function inlining is the process of replacing a call to a function by its implementation, thus reducing the overhead of the calling process. Inlining, while useful for small functions, may cause high resource usage if complex functions are inlined multiple times in the same function. The proposed heuristic for function inlining attempts to balance the size of the function being inlined with the size of the parent function, by using the number of load operations of each function as the comparison factor.

The heuristic starts by finding all distinct function calls in the function being processed. Then, the framework tries to find whether each called function has an available implementation. This is not always the case, particularly for functions implemented in external libraries and only available through a signature in a header file. Functions with no implementation available are dropped from the analysis. The function call frequency is then calculated for each valid function (including the number of times the function call is executed when inside a loop). Then, the DFG for each called function is built, and the total number of load operations is calculated by finding all the DFG nodes of the "Load" type, multiplying each occurrence by its number of iterations (calculated similarly to the function call frequency) and then summing them all. This gives an estimate of the number of load operations of each called function, called *functionCost*. Lastly, an estimation of the number of load operations is also calculated for the parent function, called *mainCost*. The function is inlined if the conditional expression in Eq. 4.1 evaluates to true.

```
1  int f1(int x[128]) {
2    int y = 0;
3    for (int i=0; i < 128; i++)
4      y = y * x[i];
5    return y;
6  }
7
8  int f2(int x) {
9    return x * x;
10 }
11
12 int inlineExample(int x[128]) {
13   int y = f2(x[0]);
14   for (int i = 0; i < 8; i++) {
15     y += x[i] + f1(x);
16   }
17   return sqrt(y);
18 }
```

Listing 4.1: Example of a source code with calls to functions that can be inlined

$$mainCost > (call\_frequency * functionCost)/N \tag{4.1}$$

N is a configurable parameter with a default value of $N = 2$. This heuristic attempts to balance the number of times the function is inlined while assuming complete loop-level parallelism (i.e., there is one distinct copy of the function for each observed call) against the cost of the original function. A larger N will allow for larger functions, as well as more frequent functions, to be declared as being inlinable, while a smaller N will restrict the cases in which it is possible to inline a function. If this heuristic evaluates to true, the function is declared as being inlinable by using the Vivado HLS directive *HLS inline*.

As an example, let us consider the functions in Listing 4.1. The function "inlineExample" is the one being processed, and it has calls to three different functions: "f1", "f2" and "sqrt". The latter is excluded, since its implementation is not available. The "inlineExample" function has a cost of 9. Given that "f1" has a cost of 128 and a call frequency of 8, the heuristic will evaluate to false, as $9 > (8 * 128)/2$, and so the function will not be inlined. The "f2" function, however, has a cost of 0, given that it has no array accesses, and has a frequency of 1. The heuristic will evaluate to true, as $9 > (1 * 0)/2$, and the function will be inlined.

4.1.2 Loop Unrolling

Loop unrolling, also known as loop unwinding, [27] is the process of reducing the number of iterations of a loop by duplicating the loop body by a determined number of times in order to reduce the number of iterations. Unrolling can be partial or complete. Partial loop unrolling is defined by an unrolling factor F, which represents the number of times the loop body is duplicated.

```
1: for every loop nest do
2:     currentLoop ← GETINNERMOSTLOOP()
3:     INSERTUNROLLINGDIRECTIVE(currentLoop)
4:     while current loop is unrolled and has an enclosing loop do
5:         enclosingLoop ← GETENCLOSINGLOOP(currentLoop)
6:         NC ← GETITERATIONS(currentLoop)
7:         NE ← GETITERATIONS(enclosingLoop)
8:         if NC = NE ∧ NC <= 4 then
9:             INSERTUNROLLINGDIRECTIVE(enclosingLoop)
10:            currentLoop ← enclosingLoop
11:        end if
12:    end while
13: end for
```

Figure 4.2: Algorithm for loop unrolling

For a loop with N iterations, the number of iterations N' upon unrolling the loop by the factor F is given by N / F. Complete loop unrolling eliminates the loop entirely by duplicating the loop body a total of N times (i.e., F = N).

The heuristic proposed to unroll a loop is described in Figure 4.2. It starts by finding all innermost loops, which on the DFG are represented by "Loop" nodes with no outward edges for other "Loop" nodes. These are, by default, fully unrolled. Then, if the loop that encapsulates the nested loop exists, and if its number of iterations is different from the number of iterations of the nested loop, the process stops. However, if its number of iterations is the same as the nested loop, and if that number is less or equal to 4, then the encapsulating loop is also unrolled. Unrolling is performed by using the Vivado HLS directive *HLS unroll*, which allows for a configurable unrolling factor to be specified.

4.1.3 Pipelining Code Regions

By default, data is consumed on a sequential manner, e.g., one array element is processed per loop iteration, and the next element may only be processed once the previous one finishes being processed. While this execution model is typical on a CPU, it is not appropriate for an FPGA, since loop-level and instruction-level parallelism can be exploited in order to process multiple elements concurrently. The initiation interval (II) of a function or loop is the minimum number of clock cycles required for it to start processing a new input [23].

On Vivado HLS, pipelining can be done by inserting the *HLS pipeline* directive [23]. It can be applied to a loop or to a function. The desired II can be manually specified, and Vivado HLS will try to generate a design with an II as close as possible to it. If no II is specified, an II value of 1 is assumed by default. This is the approach the proposed heuristic will take. A naïve way of pipelining a kernel, or a set of kernels, would be to pipeline every function. This is, however, not ideal, since the hardware resources necessary for that could easily exceed the ones provided by the target FPGA. Instead, the proposed heuristic always pipelines the loop that encloses an unrolled

nested loop. If the unrolled loop is already at the uppermost scope of the function, it pipelines the function instead.

```
1  int DSP_autocor(short ac[16], short sd[48]) {
2     #pragma HLS array_partition variable=ac complete
3     #pragma HLS array_partition variable=sd complete
4     int i;
5     int k;
6     int sum;
7     for(i = 0; i < 16; i++) {
8        #pragma HLS pipeline
9        sum = 0;
10       for(k = 0; k < 32; k++) {
11          #pragma HLS unroll
12          sum += sd[k + 16] * sd[k + 16 - i];
13       }
14       ac[i] = (sum >> 15);
15    }
16 }
```

Listing 4.2: Implementation of an autocorrelation kernel, optimized with loop unrolling, loop pipelining and array partitioning

One of the issues that hinders the potential gains of pipelining a code region comes from the fact that FPGA BRAMs can only be accessed at most twice on a single cycle. If an array is mapped to a single memory, this may lead to a suboptimal pipelining, since the rate at which data can be read and written will be limited by this hardware restriction. This problem can be attenuated by partitioning and mapping each array to different memory units. Array partitioning in Vivado HLS can be performed by using the *HLS array_partition* directive. This directive needs to be declared once per array, and it can be configured differently for each array. The proposed heuristic to partition arrays attempts to choose between two different modes of partitioning:

- Complete partition - this type of partition completely maps all elements of an array into registers, thus eliminating the need for a BRAM. This is the best-case scenario to enable instruction-level parallelism, since the memory access bottlenecks are removed. However, Vivado HLS has the restriction that only arrays of size up to 1024 elements can undergo this kind of partitioning. Furthermore, and depending on the size of the array data type, this may lead to a resource usage that exceeds the limit provided by the target FPGA;

- Cyclic partition - this type of partition divides an array into N smaller arrays, and it distributes the elements by distributing them evenly among the partitions, e.g., for an array "a" of size 9, and with a partition factor of N = 3, the first partition would have the elements a[0], a[3] and a[6], the second one the elements a[1], a[4] and a[7] and the final one the elements a[2], a[5] and a[8].

```
1:  for each input array do
2:      SizeDT ← GETDATATYPESIZE(currentArray)        ▷ Size of the data type in bytes
3:      SizeArr ← GETARRAYSIZE(currentArray)
4:      SizeTotal ← SizeDT × SizeArr
5:      if SizeTotal <= 4096 then
6:          COMPLETEPARTITION(currentArray)
7:      else
8:          CYCLICPARTITION(currentArray, 64)
9:      end if
10: end for
```

Figure 4.3: Algorithm for array partitioning

The proposed heuristic to partitioning an array is presented in Figure 4.3. It starts by checking whether the array can be partitioned completely, and if that is not possible, it falls back into a cyclic partition. In order to check whether an array can be completely partitioned, its total size is first determined. On a one-dimensional array, this is done by multiplying the size of the array by the size of its data type (on a 2-dimensional array, the size must be given by multiplying the number of rows and columns). An upper limit is established to serve as a cutoff between what can and cannot be partitioned. This limit corresponds to an array of a 32-bit data type and with a size of 1024. For a 64-bit data type, the maximum array size would need to be 512 in order to be under this limit. If an array is beyond this limit, then the heuristic falls back into a cyclic partition with a partition factor of 64, regardless of the array. This partitioning factor is chosen as an attempt to leverage the number of partitions and the hardware resources of the FPGA, as values bigger than 64 may lead to an over-usage of resources without any performance gains. Inversely, a smaller number leads to less partitions, which may lead to a less optimal pipelining. Listing 4.2 shows a version of the Autocorrelation kernel of the DSPLIB benchmark suite [29] optimized with the aforementioned heuristics, alongside loop unrolling.

4.1.4 Unrolling and Pipelining Single Loops

Some functions can undergo a different analysis using a strategy different from the ones mentioned above. These functions must have only one loop, with no nested loops, and each array can, at most, only be loaded once and stored once per iteration. If the framework detects that the targeted function follows these restrictions, and if the user specifies that this option should be used, then the framework will disregard the other strategies and apply this one exclusively. This strategy is not applied by default, since it depends on a parameter provided by the user.

This parameter is called the load/stores factor N. Using this parameter, the framework applies three Vivado HLS directives: firstly, it partitions the input arrays using the *HLS array_partition* directive using a cyclic partition with N as the partition factor. Then, it unrolls the loop using the *HLS unroll* directive and N as the unrolling factor. Finally, it pipelines the loop using the *HLS pipeline* directive. The bigger N is, the more load/store operations will be exposed on a single

iteration, potentially increasing loop and instruction level parallelism at the expense of higher resource usage.

```
1 int DSP_dotprod(short x[128], short y[128]) {
2     int sum = 0;
3
4     for (int i = 0; i < 128; i++)
5         sum += x[i] * y[i];
6     return sum;
7 }
```

Listing 4.3: Implementation of a kernel that calculates the dot product of two vectors

This parameter is called the load/stores factor N. Using this parameter, the framework applies three Vivado HLS directives: firstly, it partitions the input arrays using the *HLS array_partition* directive using a cyclic partition with N as the partition factor. Then, it unrolls the loop using the *HLS unroll* directive and N as the unrolling factor. Finally, it pipelines the loop using the *HLS pipeline* directive. The bigger N is, the more load/store operations will be exposed on a single iteration, potentially increasing loop and instruction level parallelism at the expense of higher resource usage.

Listing 4.3 shows the Dot Product benchmark from the DSPLIB [29] benchmark suite. This benchmark can be optimized with this heuristic, since it has only a single loop and both its input arrays, "x" and "y", have a single load operation per iteration. In Listing 4.4, it is possible to see the result of the application of this strategy with a load/stores factor of 8.

```
1  int DSP_dotprod(short x[128], short y[128]) {
2  #pragma HLS array_partition variable=x cyclic factor=8
3  #pragma HLS array_partition variable=y cyclic factor=8
4      int sum = 0;
5
6      for (int i = 0; i < 128; i++) {
7  #pragma HLS unroll factor=8
8  #pragma HLS pipeline
9          sum += x[i] * y[i];
10     }
11     return sum;
12 }
```

Listing 4.4: Optimized version of the Dot Product kernel using a load/stores factor of 8

4.1.5 Declaring Array Parameters as Streams

On Vivado HLS, all array variables are, by default, assigned a RAM interface port. However, if the data of this array is either produced or consumed on a sequential manner, it is possible to implement it as a stream of data through a FIFO. This can be done by using the *HLS stream*

directive [23]. When this directive is used, a reduction in the FPGA hardware resources should be expected, since we are freeing up the resources that would be required to implement a RAM interface and replacing them for a more efficient queue. In order to detect whether it is possible or not to declare a variable as a stream, the framework needs to examine the loads and stores performed on each variable, one at a time, and make a conservative guess about whether it is a sequential access or not. It is a conservative guess because if the index is obtained from the result of a complex expression that cannot be calculated at compile-time, it is not possible to decide, with certainty, that it is a sequential access. Listing 4.5 shows the implementation of the 2-stage *fir_K_A* [51] filter, whose parameter array "m" can be declared as a stream.

For an array to be declared as a stream, the framework performs a series of checks, which need to be passed on their entirety in order for the array to be declared as such. These checks are the the following:

1. Firstly, the array variable can only appear on a single loop. On a kernel with more than one stage, in which two sequential loops go through the array, the sequentiality of the accesses is not guaranteed, unless the loop nests work with different sequential partitions of the array (e.g., for an array with 8 positions, the first loop works with elements from 0 to 3, and the second one works with elements from 4 to 7).

2. All arrays must be verified as being either only read from or written to, even if the access pattern is sequential. While this can be detected by looking at all occurrences of the "Load Array" and "Store Array" nodes that reference the array being processed, special care must be taken to consider the implicit load operation previously described in Section 3.4.6;

3. For single-dimension arrays, the loop that determines the loop index must be the top-most loop, and the access must be done in its immediate scope (and not on another loop nested inside that one). Otherwise, the array, even if accessed sequentially, would be accessed sequentially multiple times, which is not allowed. For two-dimensional arrays, the same logic holds true: the loop that determines the row must be the top-most loop, the loop that determines the column must be a loop nested on the top loop, and the access must be performed on the nested loop's scope;

4. Finally, the expression that calculates the index must obey certain rules. The most simple example that is allowed is simply using the loop counter. However, more complex expressions are contemplated: if an array is accessed twice in the same loop body, and if the patterns are of the style of "i" and "i + 1", then this is allowed as long as the loop iterator counter increments the loop in intervals of 2. The same holds true for N accesses, as long as they follow this progression and the loop iterator is incremented appropriately.

Looking back at the *fir_K_A* example, it is possible to see that array "m" is the only array that passes all 4 checks. This two-dimension array is only accessed on one stage, is on the second level of a loop nest with the top-most loop providing the index for the row and the nested loop providing

the index for the column, and the access pattern is sequential, with only one load operation that transits always to the next element of the array upon each new iteration.

```
1  void fir_K_A(int t[100], int o[100], int a[100], int m[100][100]) {
2      #pragma HLS stream variable=m
3
4      for(int i = 0; i < 100; i++) {
5          t[i] = 0;
6          for(j = 0; j < 3; j++) {
7              t[i] = t[i] + (a[i - j + 2] >> 2);
8          }
9      }
10     for(int i = 0; i < 100; i++) {
11         o[i] = 0;
12         for(j = 0; j < 100; j++) {
13             o[i] = o[i] + m[i][j] * t[j];
14         }
15     }
16 }
```

Listing 4.5: Implementation of the *fir_K_A* filter, which contains an array declared as a stream

4.1.6 Support For Other Directives

The previous heuristics focus on restructuring code using the Vivado HLS directives with the most impact and use cases, but Vivado HLS offers many other directives besides these ones. This subsection lists all available Vivado HLS directives, and indicates whether they are or not currently supported by the framework. For the directives that are not supported, a proposal about how the framework could tackle them is provided, as well as a reflection about whether it would be advantageous to do so.

- *HLS allocation* - limits the maximum number of times a resource can be used (in the case of an IP core), or the number of times a code region should be synthesized (e.g., limit the number of times a function's implementation is repeated when inlined). The framework could be used to establish a worst-case upper limit for some resources, such as by counting all arithmetic operations of the same type (e.g., all "add" operations for integers) and using that count as the upper limit. Smaller limits would require heuristics, which could offer an estimate of the number of resources to be used in terms of potential parallelism (e.g., the "add" operations on a pipelined region could be estimated as being far less than the number of operations used without pipelining). Alternatively, the total amount of resources provided by the target FPGA could be provided, but that would require the framework to support target-specific optimizations, which, at the moment, is not contemplated;

- *HLS array_map* - maps small arrays into one larger array. The framework can already identify the size of each array, and, if it detects that the arrays are not accessed in parallel, it could combine them into one larger array in order to reduce hardware resource usage;
- *HLS array_partition* - already supported, see Section 4.1.3;
- *HLS array_reshape* - it performs the same kind of partition as the previous directive, but with larger bit-widths for the array elements, which allows for more data to be retrieved on a single clock cycle. The framework could support it as an improvement over the existing array partitioning strategy, but possibly with different heuristics;
- *HLS data_pack* - packs the data fields of a struct into a single scalar. Given that the framework does not yet support structs, this directive would only become useful after that extension is developed;
- *HLS dataflow* - this directive allows for data that is processed by different sequential function calls to be transferred between those functions more efficiently, e.g., the same array can be processed on more than one function at once, as long as the order is respected. Vivado HLS stipulates five conditions that must be followed in order for a code region to be declared as a dataflow region. These conditions are mostly related to access patterns, and as such their detection can be done on the framework by looking at the read/write operations being performed on each function;
- *HLS dependence* - allows for information about dependencies between variables to be specified, which help guide the HLS tool into achieving better pipelining results. It supports both intra-loop dependencies (i.e., between the statements of the loop body) and inter-loop dependencies (i.e., between iterations). The framework, at the moment, does not track dependencies when it comes to array accesses, and thus it is not possible to use this directive without first implementing that extension. However, this could be done by using the dependency analysis currently implemented in Clava's Autopar library [52];
- *HLS expression_balance* - expression balancing is the process of rewriting long arithmetic expressions using associative and commutative rules in order to reduce the height of the corresponding expression tree. Unbalanced expressions may cause a long chain of Register-Transfer Level (RTL) operations, which can negatively affect the performance of the synthesized design, but balanced expressions may lead to a different accuracy. This directive turns off expression balancing on a specified code region, since Vivado HLS does it by default. Detecting unbalanced expressions on the framework is trivial, since the tree-like structure of an arithmetic expression is already exposed by the DFG, and identifying unbalanced expressions is a simple as calculating and comparing the depths of an expression's branches. The data types of the variables can also be retrieved, and unbalanced expressions that use floating-point values can be singled out. It would be up to the user to specify whether they want to turn off expression balancing;

- *HLS function_instantiate* - this directive can be used to signal function arguments whose all possible values are known at compile-time. If this happens, different implementations of the function can be generated, with one implementation per possible value that the argument can take. The framework could support this by looking at all calls of each function and check, for each argument, if they are passed as constant literals. If the same argument is always a constant literal on all the function calls, then the directive could be used to signal that argument;

- *HLS inline* - already supported, see Section 4.1.1;

- *HLS interface* - specifies which protocol or hardware resource should be used to implement the function interface (i.e., the way function arguments and global variables are passed to and from the FPGA). One particular instance of this directive, *HLS stream*, is already supported. This is a complex directive with many possible configurations, and it could be implemented on the framework using many different approaches. Data type analysis and access pattern analysis could be performed in order to choose which memory sizes and array-passing methods (e.g., by reference or by copy) should be used, and user configurations could also be taken into consideration (e.g., the user could specify that they want it to be implemented using AXI interfaces, and then the framework could choose which AXI interfaces are the most appropriate for a given function);

- *HLS latency* - allows for a desired latency interval to be specified for a code region. There is no apparent way for the framework to support this without taking into consideration parameters provided by the user;

- *HLS loop_flatten* - merges nested loops into a single loop, within the provided scope of the directive. The framework can easily detect nested loops, but a heuristic would be needed in order to decide whether this directive should be applied, particularly when other directives are being applied to the same loop nest;

- *HLS loop_merge* - merges consecutive loops in the same scope. Similarly to the previous directive, the framework can easily detect these scenarios, but an heuristic would also need to be researched in order to decide whether it would be advantageous to apply this directive in the presence of others;

- *HLS loop_tripcount* - specifies the number of iterations of a loop with an unknown number of iterations at runtime. It has no impact on synthesis, and serves only to produce more detailed HLS reports. A similar functionality is already present in the framework by using the custom directive proposed in Section 3.4.4, with the exception that it allows for the framework to detect the number of iterations, rather than Vivado HLS. However, that custom directive could be trivially converted into the one used by Vivado HLS, and thus allow for more accurate reports to be generated;

- *HLS occurrence* - specifies a code region within a loop as being executed less times than the number of loop iterations (e.g., an if-statement that only executes on even-numbered iterations). The framework could detect this by analyzing the part of the DFG that contains the conditional expression and checking whether that expression has only constant values known at run-time and references to the loop iterator. If that happens, then the number of occurrences could be determined, and the directive could be applied;
- *HLS pipeline* - already supported, see Section 4.1.3;
- *HLS reset* - allows for fine-grained configuration of the FPGA's reset port. There is no apparent way of supporting this directive on the framework without relying on user parameters;
- *HLS resource* - specifies which hardware resource should be used to implement a variable or an arithmetic operation. Both variables and arithmetic operations are easily identifiable on the framework's DFG, and each element could be assigned an hardware resource based on a set of heuristics (e.g., an array with two parallel operations could be specified as being implemented as a dual-port BRAM), as well as based on parameters provided by the user (e.g., all arrays should be implemented as BRAMs);
- *HLS stable* - specifies a variable as being either only read or only written to on a dataflow region. The reads and writes of a variable are represented, on the framework's DFGs, by load and store nodes, and so the DFG for each function called in the dataflow regions would need to be built and, for each variable, the loads and stores would need to be analyzed in order to decide whether it could or not be declared as stable;
- *HLS stream* - already supported, see Section 4.1.5;
- *HLS unroll* - already supported, see Section 4.1.2;

4.2 Detecting Instrumentable Code

In order to determine whether a kernel can be instrumented by the tool proposed in section 3.3 and be handled over to the trace-based restructuring tool proposed by Ferreira and Cardoso [16], multiple checks must be performed. These checks are performed over the CFG and DFG, and try to detect code patterns that violate the restrictions of both tools. These checks are done in sequential order. The code must pass all checks, and it terminates prematurely if a check fails. They are the following:

1. The first validation step is the same as the one done for the directives-based restructuring, since it detects code patterns that are not allowed for both approaches. This includes the usage of pointers, dynamic memory, structs and custom type definitions. This is done during the building process of the DFG, since these language constructs can be identified by looking at the AST node types being used to build the DFG. If none are found, it passes on to the next stage;

2. The trace-based tool does not support conditional code execution (e.g., through if-else statements) beyond ternary statements. This can be detected on the CFG by looking for basic blocks of the "if" type, and for control path edges of the "true" or "false" types. A ternary statement does not require these kinds of edges and basic blocks, since it is considered to be a regular statement with no branching paths, and as such this analysis allows for code with ternary statements to pass on to the next stage. An alternative approach would be to use the DFG instead, where all "Multiplexer" nodes would be fetched and matched to see if they were made from a ternary statement or from a more complex conditional control structure, since it is possible to match all DFG nodes to the AST node that most closely matches the variable or structure represented by the DFG node;

3. In this stage, all function calls are examined to see if they match the ones allowed by the trace-based tool. That tool only allows for functions that serve as modifiers of the input, that is, functions with only one argument and one return value (e.g., sqrt). This can be detected on the DFG by selecting all "Function call" nodes and checking whether the number of inward edges equals 1, since there is one inward edge per argument being passed to the function. If all function calls follow this pattern, it can move on to the next stage;

4. Finally, this stage attempts to identify array access patterns that are too complex to be handled by the trace-based tool. The tool can work with sequences of array accesses whose indexes follow an arithmetic progression, and it uses the resolved values of the array access expressions obtained in runtime. Since the framework can only perform compile-time analysis, it is not possible to completely identify whether the array access patterns will resolve to indexes that follow this rule. However, an attempt to estimate whether an array access expression will follow this rule is made. By looking at all array access patterns, the framework accepts only expressions that comprise, at most, of a loop counter and constant values. These values can be both summed or subtracted (e.g., a[i + 1] or a[2 + i - 3]), but other operations, such as multiplications, are not allowed, since they are more likely to lead to geometric progressions rather than arithmetic ones.

4.3 Mathematical Functions Replacement

This strategy focuses on the replacement of function from math.h for more efficient equivalents. A statistical analysis of typical HLS benchmarks is performed in order to find out how common the calls to math.h functions are, and then, based on that analysis, multiple function substitutions are performed. Both the analysis and substitution of functions are implemented as LARA aspects, and can be applied independently from all other strategies. In fact, they can be applied to code targeting different HLS tools, and can even be performed over code selected to be instrumented and restructured using the trace-based framework proposed by Ferreira and Cardoso [16], if the user so desires.

Table 4.1: Number of times each math.h function is called in each benchmark. Functions that never appear and benchmarks with no function calls are omitted

Benchmark	atan	cos	sin	exp	log	log10	pow	sqrt	fabs
kmeans	0	0	0	0	0	0	0	0	1
knn	0	0	0	0	0	0	0	1	0
svm	0	0	0	1	0	0	0	0	0
triangles_hipotenuses	0	0	0	0	0	0	0	1	0
snu_ludcmp	0	0	0	0	0	0	0	0	1
snu_minver	0	0	0	0	0	0	0	0	2
utdsp_adpcm	0	0	0	0	0	2	2	0	4
utdsp_compress	1	1	0	0	0	0	0	0	0
utdsp_lpc	0	1	0	0	0	0	0	1	0
utdsp_spectral_estimation	2	2	1	0	0	1	0	0	0
utdsp_trellis	0	1	0	0	1	0	0	2	0
Total	3	5	1	1	1	3	2	5	8

4.3.1 Statistical Analysis of Benchmarks

In order to find which math.h functions are the most common across codebases of typical HLS code, a collection of 44 benchmarks was put together and analyzed. This collection consists of the following benchmarks:

- 13 benchmarks from the CHStone test suite [53];
- 19 benchmarks from the UTDSP test suite [30];
- 2 benchmarks from the SNU_RealTime test suite [54];
- 2 benchmarks from the DSPLIB test suite [29]
- A *filter_subband* kernel from an MPEG encoder [31];
- The HiFlipVX image processing library [55]
- The *triangles_hypotenuse* tuning benchmark [56]
- An SVM predict function from the article by Tsoutsouras et al. [25];
- The FIR kernel from the framework proposed by Ferreira and Cardoso [16];
- 3 in-house benchmarks:
  - A 2D FIR filter;
  - A k-Nearest Neighbors implementation;
  - A k-Means Clustering implementation;

In order to analyze each benchmark, a LARA aspect, *MathReport*, is implemented. This aspect starts by parsing the math.h file available in the building system in order to build a symbol table of all the functions it has, as well as the type of arguments each one accepts. This should be done dynamically, since math.h can have different implementations in different build systems. However, if there is no math.h present in the build system, the aspect falls back to a local math.h specification, obtained from the one provided by the MinGW compiler environment [57]. Then, for each benchmark, all function calls are gathered, and if the call is referring to a function present in the symbol table, and if the arguments are compatible in terms of arity, then a call to a math.h function is detected and recorded on a CSV file.

The results of this analysis are shown in Table 4.1, with some preprocessing done to omit the benchmarks with no calls to math.h. As it is possible to see, 11 out of the 44 analyzed benchmarks (25%) have one or more calls to math.h functions. However, and despite the fact that these benchmarks use only either integers or 32-bit floating point (FP) numbers (with the exception of the k-NN benchmark), they are calling the double (i.e., 64-bit FP) versions of the math.h functions, as the 32-bit FP versions of these functions usually have a -f suffix appended to them (e.g., *exp* becomes *expf*). This, then, opens up the possibility of obtaining more performance out of these functions by replacing each call to a 64-bit FP function by its 32-bit FP equivalent. Finally, it is also worth noticing that only 9 distinct math.h functions were detected on the analyzed benchmarks. Their frequencies of occurrence are also summarized in Table 4.1. One occurrence is the equivalent of one call to the function regardless of its context (i.e., a call to a function inside a loop still only counts as one occurrence, regardless of the number of iterations).

4.3.2 Float Versions of Mathematical Functions

The potential gains from replacing a call to a default 64-bit FP math.h function by its 32-bit version is two-fold. On one hand, the amount of resources consumed can be reduced, since the 32-bit FP numbers no longer need to be converted onto 64-bit versions. On the other, the 32-bit versions may use more efficient algorithms than the ones used by the 64-bit versions, thus improving the latency of the function. This gain can be demonstrated by synthesizing the motivational example on Listing 4.6 using Vivado HLS with a target clock of 10 ns. This synthesis is performed twice: once with the code handling an array of 32-bit FP numbers using the default 64-bit math.h functions, and once with the 32-bit versions of each function used instead. The results of the synthesis are in Table 4.2. The first aspect to notice is the existence of upper and lower limits for the latency values, which come from the fact that math.h functions may have different execution times for different inputs. The 32-bit version has, on average, a speedup of 2.71×in terms of latency when compared to the unoptimized version. As for the resource consumption in terms of Block RAMs (BRAM), Digital Signal Processors (DSP), Flip-Flops (FF) and Lookup Tables (LUT), all four resource categories have less usage on the optimized version, with an usage of only 17% BRAMs, 19% DSPs, 24% FFs and 37% LUTs when compared to the unoptimized version. These decreases in latency and resource usage make for a compelling argument in regards to investing in

Table 4.2: Comparison of the latency and resource usage of 64-bit and 32-bit versions of math.h functions

Version	Min. Latency	Max. Latency	BRAM	DSP	FF	LUT	Target Clock (ns)
Default	28161	30209	75	248	24141	18447	9.51
32-bit functions	10241	11265	13	48	5806	6877	9.39

the identification and replacement of calls to 64-bit FP functions using 32-bit FP arguments and return value.

```
1  #include <math.h>
2
3  #define N 256
4
5  float foo(float x[N]) {
6    //float is a 32-bit floating point type
7
8    float n1 = 0;
9
10   for (int i = 0; i < N; i++) {
11     n1 += cos(x[i]);
12     n1 += pow(x[i], 1.5f);
13     n1 += log10(x[i]);
14   }
15   return n1;
16 }
```

Listing 4.6: Motivational example of three opportunities for a 32-bit math.h function to be used

In order to determine whether each call to a math.h function could be replaced by the 32-bit version, another LARA aspect, *MathCompare*, is applied. This aspect takes a function as the input and then, for each call to a math.h function, reports on the type of arguments being passed to the function, the type of arguments that the function takes, and whether it is possible to replace the function by the 32-bit version. It also takes into consideration if a 32-bit version of the function is available on the math.h implementation being used by looking for the -f suffix and by checking the type of the arguments and the return value. All previously identified calls to a math.h function are accurately reported as valid opportunities for replacement using this aspect. After these opportunities are detected and validated, a third LARA aspect, *MathReplace*, is applied to each opportunity in order to replace it for the appropriate function call.

4.3.3 Optimizing Individual Functions

Some individual mathematical functions can be further optimized in order to improve their performance. Some codebases, like CHStone [53], do this by using lookup tables with precomputed

values (in fact, no calls to math.h functions are detected on the selected examples from that codebase). However, lookup tables are not generic to the point of being usable by any program. They usually include either only the values required by a certain program, which make it impossible to use them for a generic program, or include samples of the function domain, rather than all the values. This, for an arbitrary input value whose image is not on the lookup table, will imply an approximation, which leads to a loss of precision. However, it is still possible to optimize individual functions in order to achieve better performance than the one achieved by the math.h implementation. This section, then, proposes optimizations for the most commonly found math.h functions, as shown by the "Total" row of Table 4.1.

Power Function Substitution

A particular case of the power function can be identified when the exponent is an integer. In this case, for any x^n, we can abdicate of the call to the power function entirely and represent the power operation as a discrete sequence of n multiplications of the basis x. Another substitution that can be performed relates to the conversion of a power function into a root, e.g., $x^{0.5}$ can be turned into $\sqrt{x}$.

Square Root

The square root function was the second most frequent function found in the analyzed benchmarks. On Vivado HLS, the square root for both FP data types is implemented through an IP core, which is already highly performant. However, an even faster implementation of this function can be achieved at a the expense of a small decrease in accuracy. We added to the fast_math library a square root function based on the fast inverse square root function, made popular by its inclusion on Quake III's source code [58], as well as a 64-bit FP version.

Sine to Cosine Conversion

On a synthesizable input with calls to both sine and cosine functions, a resource-saving opportunity can be identified by using only one of those functions instead of both. Given the equivalency $sin(x) = cos(90^o - x)$, it is possible to convert all calls to a sine function into calls to a cosine function, or vice-versa. This would incur in a latency penalty due to the added subtraction that needs to be done to convert the angle, but would also free up the resources that would otherwise be used to implement the function. This transformation is not applied by default by the *MathReplace* aspect, but is available as an option.

4.4 Summary

This chapter described the proposed framework's code restructuring strategies. The strategies are mostly based on inserting Vivado HLS directives, with their locations and parameters determined by heuristics. The first strategy relates to inlining functions. For each function that can be inlined,

a cost function is applied in order to decide whether it should be inlined or not. This cost function compares the number of loads of the main function against the number of loads of the called function. Then, two strategies that combine loop unrolling and loop pipelining were described. Loop unrolling is applied completely to the innermost loop of a loop nest, and then to all the upper loops with a number of iterations equal or less than four. If there is a loop enclosing an unrolled loop, then that loop is pipelined in order to enable loop and instruction-level parallelism. This parallelism is further promoted by partitioning input arrays using two different strategies, based on their sizes and data type. Small arrays can be mapped directly into registers, while larger arrays may be partitioned into 64 partitions. Another strategy, configured by the user, is also proposed as an alternative to be applied to single loops with at most two load/store operations per array in a single iteration. This strategy uses a user-defined factor in order to unroll the loop by that factor, partition the input arrays also by that factor and pipeline the loop. Finally, the last strategy attempts to identify input arrays that are accessed sequentially, which allows for them to be declared as streams. Alongside these directive-based strategies, a strategy based on math.h functions was also described. This strategy identifies calls to mathematical functions that use doubles (i.e., 64-bit FP) and have arguments of the float type (i.e., 32-bit FP), and for each of these occurrences, it replaces the function by its 32-bit FP equivalent. Finally, some of the most commonly used functions can be replaced by alternative implementations in order to save resources or decrease latency.

Chapter 5

Experimental Results

This chapter describes the evaluation procedures performed in order to evaluate the proposed framework. It starts by describing the hardware and software tools used. Then, a global framework evaluation is done, in which all the generic code restructuring strategies are evaluated using a benchmark selection consisting of machine learning, matrix manipulation, filters and digital signal processing kernels and applications. Then, the user-configured heuristic for single loops is evaluated using different parameters, and finally the framework is compared to code with directives inserted manually by an expert in order to assess how close the automatic insertion of directives is to the manually optimized versions. A small evaluation of the validating process for code that can be instrumented is also provided.

5.1 Environment Setup

The FPGA development environment used for the evaluation is provided by Xilinx [59]. From an hardware standpoint, a Zedboard development kit [8] is used. This general-purpose board includes various I/O capabilities and a Xilinx Zynq-7000 SoC [7]. This chip contains both an ARM Cortex-A9 CPU and an Artix-7 FPGA [7], making it ideal for hybrid CPU/FPGA applications. The board can support multiple OSs, such as Linux, Android or Windows, as well as standalone applications, with the latter being the most relevant for this purpose. Vivado HLS [17] is the HLS tool chosen for the synthesis, since the framework optimizes code by inserting directives specific to this tool. It can also be configured to target the FPGA of this board in particular, and thus can be made aware of the available resource budget. Version 2019.1 is used for this evaluation.

5.2 Global Framework Evaluation

In order to evaluate the framework, a set of benchmarks were collected from multiple sources. These benchmarks can be divided into 4 categories: machine learning, matrix operations, filters

and digital signal processing kernels. Appendix B includes the source code of all these benchmarks. The machine learning benchmarks consist of two kernels: a k-Nearest Neighbors (kNN) classifier configurable with respect to number of features, and a support vector machine (SVM) predictor. The input sizes, loop structure and source of both these functions are presented in Table 5.1.

Table 5.1: Selected machine learning benchmarks

Benchmark	Loop Structure	Source	Data type	Input Size
kNN (K=3)	2 nested for-loops	In-house	32-bit FP	32 features 1,000 data points
				64 features 1,000 data points
				128 features 1,000 data points
				32 features 10,000 data points
SVM	2 nested for-loops	Tsoutsouras et al. [25]	32-bit FP	18 features 1248 support vectors

The benchmarks based on matrix operations have three kernels (see table 5.2): one based on the multiplication of two 10×10 matrices, one that applies a discrete cosine transform (DCT) over a 8×8 matrix and one that multiplies two 8×8 matrices of complex numbers (the matrices are provided as a flattened 1-dimensional array).

Table 5.2: Selected matrix operations benchmarks

Benchmark	Loop Structure	Source	Data type	Input Size
DCT	2 stages of 3 nested loops each	In-house	32-bit integer	5 matrices of size ×8
Complex Matrix Multiplication	3 nested loops	DSPLIB [29]	32-bit FP	2 arrays with 64 elements 1 array with 128 elements
Matrix Multiplication	3 nested loops	UTDSP [30]	32-bit FP	3 10×10 matrices

Three benchmarks based on filters are also used (see table 5.3): a fir2D filter, which implements a smoothing box filter over an image, a Kalman filter, which estimates a vector out of three input matrices, and the *fir_K_A* filter, which applies a filter to a 1-dimensional signal and remaps it onto an output vector using a matrix.

Finally, four benchmarks based on digital signal processing are selected (see table 5.4): an Autocorrelation kernel that calculates the correlation of a signal with a delayed version of that same signal, a kernel that calculates the dot product of two vectors, a Gouraud shader that calculates a vector of colors and a normalized Lattice Filter.

All these benchmarks are synthesized with Vivado HLS using a target clock of 10 ns, using both an unoptimized version as a baseline, and a version optimized through the automatic insertion

Table 5.3: Selected filter-based benchmarks

Benchmark	Loop Structure	Source	Data type	Input Size
fir2D	4 nested loops	In-house	8-bit integer	800×600 image 3×3 filter
Kalman	2 single loops 2 stages of 2 nested loops	In-house	32-bit integer	3 16×16 matrices Array with 16 elements Array with 4 elements
fir_K_A	2 stages of 2 nested loops	Book [51]	32-bit integer	100×100 matrix 3 arrays of size 100

of directives by the proposed framework. The load/stores user-configured heuristic is not considered in this analysis, since it is the focus of its own section. All the cases in which it could be applied were analyzed using the generic heuristics instead.

The results of each benchmark are split across two types of tables. Tables 5.6, 5.8, 5.10 and 5.12 have the set of directives applied to each benchmark, as well as their parameters. The "Unrolling" directive includes an "F" if the loop is fully unrolled, or the unroll factor otherwise. The "Pipelining" directive includes the initiation interval (II) achieved by Vivado HLS on each occurrence. The "Array Partition" directive has the partition parameters applied to each input array. If it is complete, it is represented by a "C"; otherwise, it has the partition factor. The "Stream" and "Math.h" columns include an "X" if they were applied at least once per benchmark. This table also includes the estimated clock period (ECP), the maximum clock frequency (MCF) and the latency (measured in number of cycles). Finally, the speedup is calculated by dividing the latency of the unoptimized version by the latency of the optimized one (i.e., considering that the synthesized designs run at the same clock frequency).

Tables 5.7, 5.9, 5.11 and 5.13 present information about the hardware resources used by each benchmark. They show the resource usage of Flip-Flops (FFs), Lookup Tables (LUTs), Digital Signal Processors (DSPs) and Block RAMs (BRAMs), with the latter being only present on Table 5.11. These tables first show how many times each resource is used on the optimized version when compared to the unoptimized one, followed by the percentage of usage of each resource on both versions when compared to the FPGA budget. The FPGA resource budget can be found on Table 5.5. Finally, the number of resources used for both versions is also presented.

Table 5.4: Selected digital signal processing benchmarks

Benchmark	Loop Structure	Source	Data type	Input Size
Autocorrelation	2 nested loops	DSPLIB [29]	16-bit integer	Array of size 160 Array of size 170
Dot Product	1 loop	DSPLIB [29]	32-bit integer	2 arrays of size 100
Gouraud	1 loop	DSPLIB [29]	32-bit integer	Array of size 200
Lattice Filter	1 loop with 2 nested loops	UTDSP [30]	32-bit float	4 arrays of size 64 Order factor of 32

Table 5.5: Hardware resources available on the target Xilinx Artix-7 FPGA

Part	#FF	#LUT	#BRAM	#DSP
xc7z020-clg484-1	106,400	53,200	280	220

5.2.1 Machine Learning Benchmarks

The optimizations performed over the machine learning benchmarks, as well has the latency speedup in relation to their unoptimized versions are present in Table 5.6, and the resource usage is presented in Table 5.7.

The kNN benchmark, on all its versions, is optimized using the same strategy: the innermost loop, whose number of iterations is given by the number of features, is fully unrolled, while the outermost loop, whose iterations are determined by the number of data points, is pipelined. All function calls are inlined on all versions. The first version uses 32 features and 1000 data points. The test and class arrays are mapped into registers, and the data points matrix is partitioned with a factor of 64 across the highest dimension. This is reflected on a speedup of 57.92×in relation to the unoptimized version's latency. In terms of resource usage, all resources have a smaller usage than the unoptimized version, which is of particular interest, since in all other benchmarks, the resource usage of the optimized version is higher than the unoptimized version. The two following versions still use the same number of data points, but use two different numbers of features, 64 and 128, in order to assess the impact of different feature sizes. The 64 feature version has a smaller speedup gain than the 32 features version, with a speedup of 32.39×in relation to its unoptimized version. Resource usage is particularly high on LUTs, of which 83.37% of the total available units are used. This version also achieved a worse pipelining result than the 32 features version (II=32 and II=9, respectively). This is due to the inner loop being fully unrolled, as the inner loop depends on the number of features. The version with 128 features, however, has a very small speedup of only 1.07×. This is due to Vivado HLS not being able to pipeline the loop, as it considers the control flow in the loop body to be too complex. This control flow comes from the inner loop, which is

Table 5.6: Optimizations and latency speedup of the machine learning benchmarks

Benchmark	Unrolling	Pipelining	Array Part.	Inlining	Stream	Math.h	Speedup	ECP (ns)	MCF (MHz)	Latency (#ccs)
kNN 32x1000	F	II=9	C,64,C	X	–	–	57.92	8.44	118.46	9237
	–	–	–	–	–	–	–	8.40	119.12	535010
kNN 64x1000	F	II=32	C,C	X	–	–	32.39	8.37	119.45	32324
	–	–	–	–	–	–	–	8.40	119.12	1047010
kNN 128x1000	F	*	C,C	X	–	–	1.07	8.90	112.36	1936010
	–	–	–	–	–	–	–	8.40	119.05	2071010
kNN 32x10000	F	II=16	C, 64	X	–	–	33.40	8.40	119.05	160180
	–	–	–	–	–	–	–	8.40	119.05	5350010
SVM	F	II=13	C, 64	–	X	X	24.85	8.40	119.05	16375
	–	–	–	–	–	–	–	8.23	121.48	406849

F full unroll, *II* initiation interval, *C* complete partition, *ECP* estimated clock period, *MCF* maximum clock frequency

Table 5.7: Resource usage of the machine learning benchmarks

Benchmark	FF X	LUT X	DSP X	%FF	%LUT	%DSP	#FF	#LUT	#DSP
kNN 32x1000	0.17	0.37	0.18	1.46	5.90	2.27	1558	3137	5
	–	–	–	8.78	15.91	12.73	9342	8462	28
kNN 64x1000	6.85	14.13	2.80	10.04	83.37	6.36	10685	44352	14
	–	–	–	1.47	5.90	2.27	1560	3138	5
kNN 128x1000	25.69	25.30	1.00	37.72	149.40	2.27	40132	79480	5
	–	–	–	1.47	5.90	2.27	1562	3141	5
kNN 32x10000	4.43	8.33	2.80	6.56	49.23	6.36	6978	26193	14
	–	–	–	1.48	5.91	2.27	1574	3146	5
SVM	1.17	2.30	1.16	4.27	30.58	23.64	4540	16270	52
	–	–	–	3.65	13.27	20.45	3886	7062	45

fully unrolled, and with 128 features, this would create a loop body too complex to be pipelined. A kNN version with a linear distance function, rather than quadratic, is synthesized in order to try to reduce the complexity of the control flow, but it had no effect. Finally, a kNN version with 32 features and 10000 data points instead of 1000 is also synthesized in order to assess the impact of the number of data points. The number of data points defines the number of iterations of the outer loop, which is pipelined with an II of 16. This leads to a speedup of 33.40×when compared to the unoptimized version. Although this result is very similar to the version with 64 features and 1000 data points, this version requires significantly less LUTs, with an usage of 49.23%. In summary, the number of features are a more limiting factor than the number of data points, given that the number of features directly impact the size of an unrolled loop.

The SVM benchmark has its test vector, with size 18, be completely partitioned into registers, while each support vector is partitioned with a factor of 64. There is also a vector with the coefficient of each support vector, which is implemented as a stream. Finally, the call to the exponent function is replaced by its float equivalent, given that this SVM benchmark uses 32-bit floating-point numbers. This optimized version has a speedup of 24.85×when compared to the unoptimized version. The outermost loop is pipelined, and Vivado HLS manages to achieve a II of 13. This value can be explained by an inter-loop dependency on a counter that is incremented on each loop iteration. In terms of resource usage, the optimized version uses resources within the FPGA limits, with the highest value being an usage of 30.58% of all the LUTs available. It is also very close to the unoptimized version in terms of resources required, as it uses only 2.30×more LUTs than that version.

5.2.2 Matrix Manipulation Benchmarks

The optimizations performed over the matrix-based benchmarks, as well has the latency speedup in relation to their unoptimized versions are presented in Table 5.8, and the resource usage is presented in Table 5.9.

The framework optimizes the DCT benchmark by completely partitioning all arrays into registers, due to the small size of the input. This proves to have minimal impact in resource usage,

Table 5.8: Optimizations and latency speedup of the matrix processing benchmarks

Benchmark	Unrolling	Pipelining	Array Part.	Speedup	ECP (ns)	MCF (MHz)	Latency (#ccs)
DCT	F	II=4, II=1	C,C,C,C	13.29	8.51	117.51	330
	–	–	–	–	8.51	117.51	4386
Complex Matrix Multiplication	F	II=2	C,C,C	26.51	8.72	114.65	160
	–	–	–	–	7.26	137.82	4241
Matrix Multiplication	F	II=5	C,C,C	20.15	9.17	109.05	557
	–	–	–	–	7.26	137.82	11221

F full unroll, *II* initiation interval, *C* complete partition, *ECP* estimated clock period, *MCF* maximum clock frequency

since the largest resource usage is of only 18.10% DSPs, which is due to unrolling loops rather than mapping arrays into registers. The usage of FFs and LUTs is of only 2.07% and 3.98%, respectively. The achieved speedup in relation to the unoptimized version was of 13.29×.

The complex matrix multiplication benchmark also has all its inputs and outputs completely mapped into registers. It has a speedup of 26.51×when compared to the unoptimized version, and the resource usage remains low even on this version. LUTs are the most used resource, with 18.41×of all total available LUTs being used, and it uses 4.13×more LUTs than the unoptimized version. DSPs are the second most used resource, with a usage of 18.18×, which is a small increase of 2.50×when compared to the unoptimized version. Vivado HLS can achieve a II of 2, which is expected since there are only inter-iteration dependencies on the innermost loop, which is completely unrolled.

The matrix multiplication benchmark, similarly to the others, also has all the three matrices partitioned into registers. Resource usage remains very low, with the most demanding resource being LUTs, with an usage of only 5.5%. However, the biggest increase is in FFs, since it requires 5.25×more units than the unoptimized versions. The speedup in relation to the unoptimized version is of 20.15×.

In summary, complex matrix multiplication benchmark achieved a better speedup than the simple matrix multiplication, despite having more operations and non-trivial array access patterns. This could be partially explained by the II achieved by Vivado HLS, which is 2 on complex and 5 on the simple one. The DCT benchmark has the least gain, with a speedup of only 13.29×when

Table 5.9: Resource usage of the matrix processing benchmarks

Benchmark	FF X	LUT X	DSP X	%FF	%LUT	%DSP	#FF	#LUT	#DSP
DCT	6.68	4.19	8.00	2.07	3.98	18.18	2199	2119	40
	–	–	–	0.31	0.95	2.27	329	506	5
Complex Matrix Multiplication	3.49	4.13	2.50	4.63	18.41	18.18	4921	9792	40
	–	–	–	1.33	4.45	7.27	1410	2369	16
Matrix Multiplication	5.25	2.87	2.00	2.62	5.05	4.55	2791	2689	10
	–	–	–	0.50	1.76	2.27	532	936	5

Table 5.10: Optimizations and latency speedup of the filter-based benchmarks

Benchmark	Unrolling	Pipelining	Array Part.	Stream	Speedup	ECP (ns)	MCF (MHz)	Latency (#ccs)
fir2D	F	II=5	64, 64	–	7.00	8.45	118.40	2386045
	–	–	–	–	–	8.11	123.29	16703337
Kalman	F	II=12, II=1	C,C,C,C,C	–	6.22	8.74	114.39	218
	–	–	–	–	–	8.51	117.51	1356
fir_K_A	F	II=2, II=100	C,C,C	X	7.53	8.74	114.39	10210
	–	–	–	–	–	8.51	117.51	41302

F full unroll, *II* initiation interval, *C* complete partition, *ECP* estimated clock period, *MCF* maximum clock frequency

compared to 26.51×and 20.15×, but it also has 2 nests of 3 loops, while the other two have only 1 nest of 3 loops each. Loop nests are not merged, so one loop nest can only start being executed when the previous one finishes, which is a limitation that could be handled in the future by adding support for loop merging, whether indirectly through directives or directly through code restructuring.

5.2.3 Filter-based Benchmarks

The optimizations performed over the filter-based benchmarks, as well has the latency speedup in relation to their unoptimized versions are presented in Table 5.10, and the resource usage is presented in Table 5.11.

The fir2D benchmark, due to its sizable input size (an array representing a 800×600 image), has its input partitioned with a factor of 64. The smoothing kernel, however, is mapped into registers, since it is a 3×3 matrix. The smoothing kernel is operated through on the two innermost loops, which are completely unrolled, and the enclosing loop is pipelined. This leads to a speedup of 7.00×when compared to the unoptimized version. The fir2D bechmark has complex array access patterns, with long expressions, and has casts between 16-bit and 8-bit integers. This, coupled with the partitioning of a 800×600 matrix into 64 individual partitions, leads to high values of resource usage. The optimized version uses 64.17% of all available FFs and 79.45% of all available LUTs. This leads to an increase of 394.69×more FFs and 133.33×more LUTs than the unoptimized version, but all values are still within the FPGA limits.

Table 5.11: Resource usage of the filter-based benchmarks

Benchmark	FF X	LUT X	BRAM X	DSP X	%FF	%LUT	%BRAM	%DSP	#FF	#LUT	#BRAM	#DSP
fir2D	394.69	133.33	1.00	5.00	64.17	79.45	0.00	4.55	68282	42266	0	10
	–	–	–	–	0.16	0.60	0.00	0.91	173	317	0	2
Kalman	8.44	8.80	2.00	11.25	3.84	9.79	0.71	61.36	4087	5209	2	135
	–	–	–	–	0.45	1.11	0.00	5.45	484	592	0	12
fir_K_A	30.63	19.43	1.00	99.00	7.63	14.68	0.00	135.00	8118	7811	0	297
	–	–	–	–	0.25	0.76	0.00	1.36	265	402	0	3

Table 5.12: Optimizations and latency speedup of the digital signal processing benchmarks

Benchmark	Unrolling	Pipelining	Array Part.	Stream	Speedup	ECP (ns)	MCF (MHz)	Latency (#ccs)
Autocorrelation	F	II=2	C,C	–	31.23	10.72	93.27	164
	–	–	–	–	–	6.38	156.74	5121
Dot Product	F	–	–	X	2.95	10.75	93.01	102
	–	–	–	–	–	6.38	156.74	301
Gouraud	F	–	–	–	3.05	7.66	130.62	66
	–	–	–	–	–	5.81	172.24	201
Lattice Filter	F	II=141	C,C,C	X	5.07	10.28	97.24	9184
	–	–	–	–	–	7.26	137.82	46593

F full unroll, *II* initiation interval, *C* complete partition, *ECP* estimated clock period, *MCF* maximum clock frequency

The Kalman filter benchmark has all its arguments mapped into registers, due to their small size. The two initialization loops are completely unrolled, and then the two loop nests get the innermost loop completely unrolled and the outermost loop pipelined. Vivado HLS achieves a II of 12 for the first loop nest, and 1 for the second. This may be due to the arithmetic expression being calculated on the latter being simpler than the arithmetic expression present on the former. This optimized version has a speedup of 11.25×when compared to the unoptimized version. In terms of resource usage, the most used resource, by far, is DSPs, with an usage of 61.36%, which is 11.25×more than the unoptimized version. This is due to the existence of complex arithmetic operations in the body of the unrolled loops. It is also worth noticing the usage of 2 BRAMs on the optimized version, which has not been observed in any other benchmarks.

The *fir_K_A* benchmark has all its input arrays mapped into registers, with the exception of the input matrix "m", which was declared as a stream due to it being accessed sequentially. Each of the two loop nests follows the pattern of completely unrolling the innermost loop and pipelining the outermost one. However, in this example, one of the innermost loops has 100 iterations, which leads to a very high II value of 100 and a DSP demand of 135%, which exceeds the resources provided by the FPGA. The loop with 100 iterations has both a multiplication and two additions on its body, and if this body is repeated 100 times, this leads to many operations that need to be implemented with DSPs.

To summarize, the filter-based benchmarks have achieved moderate improvements in speedup, with small resource usage across the board in most cases. However, a limitation with completely unrolling loops with many iterations and with resource-intensive operations in their bodies can be identified, as these scenarios may lead to excessive resource usage.

5.2.4 Digital Signal Processing Benchmarks

The optimizations performed over the digital signal processing benchmarks, as well has the latency speedup in relation to their unoptimized versions are presented in Table 5.12, and the resource usage is presented in Table 5.13.

Table 5.13: Resource usage of the digital signal processing benchmarks

Benchmark	FF X	LUT X	DSP X	%FF	%LUT	%DSP	#FF	#LUT	#DSP
Autocorrelation	8.07	95.69	10.00	0.77	24.82	4.55	815	13205	10
	–	–	–	0.09	0.26	0.45	101	138	1
Dot Product	34.43	23.21	100.00	2.65	3.10	45.45	2823	1648	100
	–	–	–	0.08	0.13	0.45	82	71	1
Gouraud	76.63	145.64	1.00	9.58	62.42	0.00	10192	33205	0
	–	–	–	0.13	0.43	0.00	133	228	0
Lattice Filter	5.81	2.89	1.88	7.89	13.49	13.64	8391	7177	30
	–	–	–	1.36	4.67	7.27	1443	2484	16

The Autocorrelation achieves a speedup of 31.23×in relation to the unoptimized version. Both inputs are partitioned completely into arrays, and the outer loop can be pipelined with an II of 2. The resource usage of the optimized version remains low. LUTs are the most used resource, with an utilization of 24.82%, which is 95.69×more than the unoptimized version. This is mostly due to the mapping of the input arrays into registers.

The Dot Product benchmark has a speedup of only 2.95×. Since it has only one loop, that loop is completely unrolled with no pipelining, and both input arrays are modeled as streams since they are accessed sequentially. These results show that the generic heuristics for directive insertion are not fitting for such a small example. The application of the load/stores heuristic for this kind of benchmark, as previously mentioned, is not being performed on this evaluation, as it will be the topic of its own evaluation process in Section 5.3. Similarly to the Dot Product benchmark, the optimized Gouraud benchmark also has a small speedup of 3.05×in relation to the unoptimized version. This is another benchmark with a single loop and only one array access per iteration, and these results are further evidence that the generic heuristics are not suitable for these simple examples.

Finally, the Lattice Filter benchmark had all its input arrays mapped into registers, except for the "data" array, which is declared as a stream. Its two innermost loops are fully unrolled, and the outermost loop is pipelined. This pipelining, however, can only be performed with an II of 141, which leads to a modest speedup of 5.07×. The pipelined loop has a very complex control flow, given that the two unrolled innermost loops have complex arithmetic expressions. This, alongside unusual array access patterns, may explain this high II value.

5.3 User-specified Load/Stores Parameter for Single Loops

In order to evaluate the user-configured heuristic used for the particular case in which there is a single loop with at most 2 load/store operations per array in a single iteration, 4 benchmarks in which this heuristic can be applied are selected. Two of these benchmarks, Dot Product and Gouraud, are also used on the global framework evaluation. The other two are functions extracted and adapted from the Rosetta benchmark suite [60]. The first of these two new benchmark functions, *computeGradient*, multiplies each entry of the input array by a scalar and stores the result on

Table 5.14: Selected benchmarks to evaluate the load/stores user-defined heuristic

Benchmark	Load/Stores of each array per iteration	Source	Data type
Dot Product	x: 1 load y: 1 load	DSPLIB [29]	16-bit integer
Gouraud	p: 1 store	DSPLIB [29]	32-bit integer
computeGradient	feature: 1 load grad: 1 store	Rosetta [60]	32-bit FP
updateParameter	grad: 1 load feature: 1 load, 1 store	Rosetta [60]	32-bit FP

a new array. The second, *updateParameter*, adds to an already initialized array position the result of the multiplication of a variable and an element from another array.

Multiple versions of the same benchmark are generated: 4 versions using a different number for the load/stores factor N (2, 8, 16 and 32), and one using the same heuristic used for the generic cases. An input size of 2000 is used for all input arrays. These benchmarks, together with the number of load/stores per iteration in each version, are presented in Table 5.14. The speedup of each version of all benchmarks, as well as the optimizations performed, are gathered in Table 5.15, and the resource usage of each version is presented on Table 5.16. The load/stores value used for each benchmark is given by the "L/S" column, where "G" represents the generic heuristic and "–" the unoptimized version.

On a global scale, all four benchmarks had a speedup increase in relation to the unoptimized version by simply using the generic heuristic, with speedups of, respectively, 2.99×, 2.00×, 6.99×and 11.95×. However, these speedup values are not very high, and for three out of the four benchmarks, the amount of hardware resources required far exceed the ones provided by the target FPGA. The Dot Product benchmark, for instance, uses 909% of all available DSPs, while the Gouraud benchmark uses 913.91% of all FFs. The *updateParameter* benchmark is not as resource demanding as the previous two, but it still uses 203.83% of all available FFs. This unrealistic resource usage and the modest gains in speedup reveal, then, that the generic heuristic is not appropriate for this kind of code patterns. For all benchmarks, the versions using the load/stores heuristic have, for all cases, better or equal speedup to the generic version, with no examples exceeding the resource budget of the FPGA.

Focusing now on each specific benchmark, it can be observed that the latency speedup of the Dot Product benchmark increases with the load/stores factor, achieving the best speedup result, 46.16×, when a load/stores factor of 16 is used. However, the latency speedup decreases to 32.26×when using a factor of 32. This might be caused by a suboptimal pipelining, since Vivado HLS only manages to pipeline that version with an II of 3 instead of the optimal pipelining of 1 that it achieved on the benchmarks with a smaller factor. In terms of resource usage, all versions use more resources than the unoptimized version, but never exceed the resources provided by the FPGA. The most percentage of usage, 14.55%, happens in the number of DSPs required for the version with a factor of 32. It is possible to observe that the number of DSPs used always equals

Table 5.15: Optimizations and latency speedup of the benchmarks using different load/store values

Benchmark	L/S	Unrolling	Pipelining	Array Part.	Stream	Speedup	ECP (ns)	MCF (MHz)	Latency (#ccs)
Dot Product	32	32	II=3	32, 32	–	31.26	9.40	106.38	192
	16	16	II=1	16, 16	–	46.16	8.93	111.98	130
	8	8	II=1	8, 8	–	23.53	8.93	111.98	255
	2	2	II=1	2, 2	–	5.98	8.93	111.98	1004
	G	F	–	–	X	2.99	10.75	93.02	2005
	–	–	–	–	–	–	6.38	156.74	6001
Gouraud	32	32	II=11	32	–	2.90	8.36	119.65	689
	16	16	II=6	16	–	2.66	8.36	119.65	752
	8	8	II=3	8	–	2.66	8.36	119.65	752
	2	2	II=1	2	–	2.00	8.36	119.65	1002
	G	F	–	–	–	2.00	8.36	119.65	999
	–	–	–	–	–	–	5.81	172.12	2001
computeGradient	32	32	II=1	32, 32	–	200.01	5.70	175.44	70
	16	16	II=1	16, 16	–	106.07	5.70	175.44	132
	8	8	II=1	8, 8	–	54.48	5.70	175.44	257
	2	2	II=1	2, 2	–	13.90	5.70	175.44	1007
	G	F	–	64	X	6.99	9.00	111.11	2004
	–	–	–	–	–	–	5.70	175.44	14001
updateParameter	32	32	II=1	32, 32	–	320.01	7.26	137.74	75
	16	16	II=1	16, 16	–	175.19	7.26	137.74	137
	8	8	II=1	8, 8	–	91.61	7.26	137.74	262
	2	2	II=1	2, 2	–	23.72	7.26	137.74	1012
	G	F	–	64	X	11.95	10.46	95.60	2009
	–	–	–	–	–	–	7.26	137.74	24001

F full unroll, *II* initiation interval, *ECP* estimated clock period, *MCF* maximum clock frequency

Table 5.16: Resource usage of the benchmarks using different load/store values

Benchmark	L/S	FF X	LUT X	DSP X	%FF	%LUT	%DSP	#FF	#LUT	#DSP
Dot Product	32	18.54	8.42	32.00	1.57	1.12	14.55	1669	598	32
	16	12.03	5.48	16.00	1.02	0.73	7.27	1083	389	16
	8	6.36	0.45	8.00	0.54	0.06	3.64	572	32	8
	2	2.46	2.23	2.00	0.21	0.30	0.91	221	158	2
	G	788.51	499.52	2000.00	66.70	66.67	909.09	70966	35466	2000
	–	–	–	–	0.08	0.13	0.45	90	71	1
Gouraud	32	12.35	21.86	0.00	1.58	9.37	0.00	1679	4984	0
	16	6.54	11.06	0.00	0.84	4.74	0.00	889	2522	0
	8	3.33	5.71	0.00	0.43	2.45	0.00	453	1303	0
	2	1.01	1.69	0.00	0.13	0.72	0.00	137	385	0
	G	913.91	1372.65	0.00	116.82	588.28	0.00	124292	312965	0
	–	–	–	–	0.13	0.43	0.00	136	228	0
computeGradient	32	27.34	26.04	32.00	6.37	19.58	43.64	6780	10414	96
	16	13.99	13.20	16.00	3.26	9.92	21.82	3469	5278	48
	8	7.31	6.78	8.00	1.70	5.09	10.91	1814	2710	24
	2	2.31	1.96	2.00	0.54	1.47	2.73	574	784	6
	G	10.14	30.60	2.00	2.36	23.00	2.73	2515	12238	6
	–	–	–	–	0.23	0.75	1.36	248	400	3
updateParameter	32	33.78	29.71	32.00	16.57	44.96	72.73	17633	23918	160
	16	17.08	14.94	16.00	8.38	22.61	36.36	8914	12030	80
	8	8.71	7.56	8.00	4.27	11.44	18.18	4547	6086	40
	2	2.28	1.98	2.00	1.12	3.00	4.55	1191	1596	10
	G	203.83	65.71	17.00	100.00	99.43	38.64	106400	52896	85
	-	–	–	–	0.49	1.51	2.27	522	805	5

the load/stores factor. This comes from the fact that there is only one operation that requires a DSP (the multiplication between the two array elements), and as such, for an unrolling factor N, there is a need for N DSPs to be allocated for the loop body.

The Gouraud benchmark shows the least gains out of the four benchmarks. The speedup for a factor of 2, 2.00×, is the same as the one achieved by the generic heuristic. The speedup increases with factors 8 and 16, although it is the same on both those scenarios (2.66×). With a factor of 32, a speedup of 2.90×is achieved. The most likely reason for these results stem from the fact that there are inter-iterations dependencies, which lead to suboptimal pipelining (the II achieved by Vivado HLS increases with each higher load/stores factor). The Gouraud benchmark fills an array with RGB color values in a way that, for each new color, its value is calculated from the previous one. This leads to inter-iteration dependencies, since the color values cannot be calculated independently of each other.

The *computeGradient* benchmark shows a consistent speedup increase with each bigger load-/stores factor, with a speedup of 200×for a factor of 32. This is expected, since Vivado HLS manages to always achieve a II of 1 in all cases. The resource usage is, once again, dominated by the DSPs, of which 43.64% of the available units are used. The increase in DSPs can also directly correlate to the load/stores factor. Three DSPs are required to implement the original loop body, and so by unrolling the loop by a factor of N, 3×N DSPs are required.

The *updateParameter* benchmark has results similar to the previous one, as the speedup increases with the increase of the load/stores factor, with a speedup of 320×for a factor of 32, and

with an achieved II of 1 for all cases. However, resource usage is higher, since there is one extra operation being performed in comparison to the *computeGradient* benchmark. Instead of the result of the multiplication being stored on the array position, this result is added to an already existing array position, which implies an additional load operation of the existing value and an addition. This can be observed on the resource usage, which starts to use higher percentages of FFs (16.57%) and LUTs (44.96%) being used for the version with a factor of 32. The number of DSPs is, once again, directly correlated to the unrolling factor: 5 DSPs are required by the original loop body, and so, for a factor of N, 5×N DSPs are required.

5.4 Comparison to Manual Code Restructuring

In order to evaluate how the quality of the automatic directive insertion compares to those manually inserted by an experienced user, five examples of manual optimization are selected. Four of these examples were previously used in the validation of the tool proposed by Ferreira and Cardoso [16], and three of these benchmarks, Dot Product, Autocorrelation and SVM, were also already used for the global framework evaluation. The Autocorrelation and SVM benchmarks use the same input sizes as the ones used for the global framework evaluation. The Dot Product benchmark uses an input size of 2000 instead of 100. An extra benchmark, *filter_subband*, is also used. This benchmark is already detailed in Section 2.2.3 as an example of code restructured by the tool proposed by the authors. It includes two nests of two loops, and three input arrays with a 64-bit FP datatype of size 512, 32 and 1024. Finally, the fifth benchmark, *computeGradient*, is extracted from the Rosetta benchmark suite [60] and was previously used to evaluate the load/stores heuristic. For this evaluation, and input size of 1024 is chosen in order to preserve the original value used in the benchmark suite. Three different versions of each of these five benchmarks is synthesized using Vivado HLS: one unoptimized, one optimized with automatic directive insertion by the framework, and one optimized manually by the authors. The *computeGradient* benchmark is an exception, since it has two automatically optimized version: auto32, with a load/stores factor of 32, and auto64, with a load/stores factor of 64. Similarly to the previous evaluations, the results have been split across two tables: Table 5.17 presents the optimizations performed on each version and the achieved latency and speedup in relation to the unoptimized version, while Table 5.18 presents the resource usage of each synthesized version.

The Autocorrelation benchmark has very similar results on both the automatic and manual versions, with a speedup of 31.23×and 31.04×, respectively. The manual version maps the "sd" array into registers, and pipelines the outermost loop, while the automatic version maps both input arrays into registers, pipelines the outermost loop and completely unrolls the innermost one. The manual version achieved an II of 1, while the automatic version achieved an II of 2. In terms of resource usage, the manual version has slightly higher usage, but not to the point of being significant (e.g., the manual version uses 95.92×more LUTs than the unoptimized version, while the automatic version uses 95.69×more).

Table 5.17: Optimizations and latency speedup of the benchmarks optimized manually and automatically

Benchmark	Unrolling	Pipelining	Array Part.	Stream	Math.h	Speedup	ECP (ns)	MCF (MHz)	Latency (#ccs)
Autocorrelation - auto	F	II=1	C,C	–	–	31.23	10.72	93.28	164
Autocorrelation- manual	–	II=1	C	–	–	31.04	10.72	93.28	165
Autocorrelation - none	–	–	–	–	–	–	6.38	156.74	5121
Dot Product - auto	16	II=3	16,16	–	–	46.16	8.93	114.68	130
Dot Product - manual	F	II=20	50,50	–	–	240.04	8.72	93.02	33
Dot Product - none	–	–	–	–	–	–	6.38	156.74	6001
filter_subband - auto	F,F	II=1, II=1	C,C,64	–	–	65.03	8.20	121.95	499
filter_subband - manual	4,4	II=8, II=49	2,2,2	–	–	37.21	10.49	95.33	872
filter_subband - none	–	–	–	–	–	–	8.23	121.51	32450
SVM - auto	F	II=13	C, 64	X	X	24.85	8.40	119.05	16375
SVM - manual	–	II=13	–	–	–	24.84	8.40	119.05	16382
SVM - none	–	–	–	–	–	–	8.23	121.51	406849
computeGradient - auto64	64	II=1	64, 64	–	–	341.38	8.02	124.69	21
computeGradient - auto32	32	II=1	32, 32	–	–	183.82	5.70	175.44	39
computeGradient - manual	32	II=1	32, 32	–	–	199.14	8.51	117.51	36
computeGradient - none	–	II=1	–	–	–	–	5.70	175.38	7169

F full unroll, *II* initiation interval, *C* complete partition, *ECP* estimated clock period, *MCF* maximum clock frequency

Table 5.18: Resource usage of the benchmarks optimized manually and automatically

Benchmark	FF X	LUT X	BRAM X	DSP X	%FF	%LUT	%BRAM	%DSP
Autocorrelation - auto	8.07	95.69	1.00	10.00	0.77	24.82	0.00	4.55
Autocorrelation - manual	8.87	95.92	1.00	10.00	0.84	24.88	0.00	4.55
Autocorrelation - none	–	–	–	–	0.09	0.26	0.00	0.45
Dot Product - auto	12.03	5.48	1.00	16.00	1.02	0.73	0.00	7.27
Dot Product - manual	402.02	1577.55	1.00	220.0	34.01	210.54	0.00	100.00
Dot Product - none	–	–	–	–	0.08	0.13	0.00	0.45
filter_subband - auto	56.32	70.25	3.00	64.00	69.07	277.69	2.14	407.27
filter_subband - manual	26.77	8.03	162.00	6.00	32.83	31.76	115.71	38.18
filter_subband - none	–	–	–	–	1.23	3.95	0.71	6.36
SVM - auto	1.31	2.76	1.00	1.16	4.27	30.58	0.00	23.64
SVM - manual	2.13	1.58	1.00	1.16	6.92	17.52	0.00	23.64
SVM - none	–	–	–	–	3.26	11.09	0.00	20.45
computeGradient - auto64	37.52	51.72	1.00	64.00	8.75	38.88	0.00	87.27
computeGradient - auto32	27.34	26.04	1.00	32.00	6.37	19.58	0.00	43.64
computeGradient - manual	6.97	1.96	1.00	21.33	1.62	1.47	0.00	29.09
computeGradient - none	–	–	–	–	0.23	0.75	0.00	1.36

The manual version of the Dot Product benchmark partitions the input arrays using a factor of 50 and using the "block" strategy, alongside pipelining the function. This leads to a speedup value of 240×. However, this high speedup comes at the expense of excessive resource usage, with an usage of all available DSPs (due to the specification of a directive that limits the overusage of this resource type) and an usage of 210.54% of LUTs, which falls beyond the resources provided by the target FPGA. The automatic version, however, is optimized using the load/stores heuristic with a factor of 16, and has a speedup of 46.16×. In terms of resources, it is well within the bounds, with the highest usage being of only 1.02% of FFs.

The *filter_subband* benchmark has a speedup of 65.03% on the automatic version, but at the expense of exceeding the available FPGA resources, as it has a usage of 407.27% of all available DSPs. In contrast, the manual version has a more modest speedup of 37.21%, but its resource usage is well within the FPGA limits. The manual version is restructured by partitioning the arrays with a factor of 2, plus pipelining of the outer loops with a factor of 4. The automatic version, however, partitions the largest array with a factor of 64 and maps the others into registers (since the data type is 64-bit floating point, the heuristic falls back to a cyclic partition rather than register mapping with a smaller threshold than the 32-bit versions), while pipelining the outer loops and fully unrolling the inner loops. Still, the complete unrolling of a loop with 64 iterations and arithmetic operations using 64-bit FP operands leads to this excessive resource usage.

The manual version of the SVM benchmark simply pipelines the outermost loop. This leads to an II of 13, and a speedup of 24.84×. The automatic version undergoes the same optimizations as described in the global framework evaluation: partition of the support vector matrix by a factor of 64, mapping the test vector into registers, streaming the coefficient values, pipelining the outer loop, unrolling the inner one and replacing the call to the exponent function by its float version. These optimizations lead to a speedup of 24.85×, which is very close to the one achieved manually. The manual version uses more FFs, while the automatic one uses more LUTs. Finally, the *computeGradient* benchmark achieves a very similar result in terms of latency speedup using a load/stores factor of 32, with more resource usage when compared to the manually optimized version. This higher usage is due to the manually optimized version replacing the floating-point data type by a more efficient fixed-point one, while the similarity in latency speedup is explained by both versions using a similar partitioning, unrolling and pipelining configuration with a factor of 32. This is performed using a Vivado HLS library for fixed-point arithmetic. The automatic version with a load/stores factor of 64, however, achieves a speedup of 341.38×while keeping resource usage within the FPGA limits, and thus achieving a design with a better latency result than the manual version. To summarize, three of the benchmarks, Autocorrelation, SVM and *computeGradient*, can achieve a result similar to the one achieved by manual code restructuring, with the latter achieving an even better result with a higher load/stores factor. Another benchmark, Dot Product, has less speedup on the automatic version, but this is due to excessive resource usage by the manual version. Inversely, the *filter_subband* has a better speedup on the automatic version, but at the expense of excessive resource usage.

5.5 Instrumentable Code Classification Accuracy

In order to evaluate the validation step described in Section 4.2 that decides whether a function can be instrumented and used in the trace-based approach proposed by Ferreira and Cardoso [16], 15 benchmarks, selected from the previous suites, are validated by the framework. The results of this validation are then compared to the results that come from executing the trace-based approach manually for each example, to see if the prediction matches reality. These results are presented in Table 5.19, with the respective confusion matrix presented on Table 5.20.

Table 5.19: Predicted and actual truth values for whether a function can be instrumented

Benchmark	Predicted	Actual
SVM	Yes	Yes
Kalman filter	No	No
fir_K_A	Yes	Yes
fir2D	No	Yes
filter_subband	No	Yes
DCT	Yes	No
Gouraud	Yes	Yes
Autocorrelation	Yes	Yes
Dot Product	Yes	Yes
Complex Matrix Multiplication	No	No
k-NN	No	No
Lattice Filter	Yes	No
Matrix Multiplication	Yes	Yes
updateParameter	Yes	Yes
computeGradient	Yes	Yes

With 15 examples, there is an accuracy of 73%, with a precision of 80% for the positive cases and a precision of 60% for the negative ones. This indicates that the heuristic is conservative, since it fails to validate some benchmarks that can be instrumented. The two benchmarks that are incorrectly invalidated are fir2D and *filter_subband*. These benchmarks have complex array access patterns that can only be evaluated at runtime, and as such the heuristic invalidates them as it is not possible to decide, at compile time, that the accesses will resolve to a pattern that the tool can recognize.

As for the two examples misclassified as being instrumentable, DCT and Lattice Filter, they have simple array accesses, but also have stages, which further complicates the overall array access pattern. It is also worth noticing that this collection of benchmarks does not contain any

Table 5.20: Confusion matrix of the instrumentation validator

		Predicted	
		Yes	No
Actual	Yes	8	2
	No	2	3

examples of code with if-statements and other constructs that are not allowed by the tool, since that evaluation is not subject to an heuristic and is only dependent on the implementation.

5.6 Execution Time Evaluation

In order to evaluate the execution time of the framework, the largest benchmark in terms of functions being processed, k-NN, is chosen as the input. Clava is executed a total of 10 times: 5 with the framework not being applied to the code (i.e., it parses the input source code and outputs an identical version of that code), and 5 with the framework being applied. These values are measured on an Intel Core i5-4460 CPU running at 3.2 GHz. On average, the version with no framework has an execution time of 5319 ms, while the version with the framework has an execution time of 5553 ms. This means that the framework, for the k-NN benchmark, only adds 234 ms on top of the time required for the default Clava tasks.

5.7 Summary

This chapter presented the evaluation of the framework. Firstly, the hardware and software suite was presented, which is based on Vivado HLS and a Xilinx Artix-7 FPGA. Then, a selection of benchmarks was detailed. This selection can be divided in machine learning, matrix manipulation, filters and digital signal processing. Each benchmark was synthesized using Vivado HLS with a target clock of 10ns, using both the unoptimized version and a version automatically optimized with the strategies proposed. Each benchmark was, then, further analyzed by comparing the synthesis results of the optimized version against the unoptimized one in terms of latency speedup and resource usage. The load/stores strategy was evaluated separately using four benchmarks and load/stores factors of 2, 8, 16 and 32. Following that, the framework was compared four benchmarks manually optimized with directives, in order to assess how it compared to manual code restructuring. Following that, the validator for code that can be instrumented and be processed using the trace-based approach of Ferreira and Cardoso [16] was evaluated using 15 benchmarks, and the results of the validator were compared to the actual results obtained by the tool. Finally, the execution time of Clava with the framework was measured and compared to the execution time of Clava with no other tasks.

Chapter 6

Conclusions

This chapter presents a thorough discussion of the dissertation results, followed by a suggestion of possible future improvements.

6.1 Concluding Remarks

This dissertation proposes a new way of optimizing C code for FPGAs on the source-to-source compiler Clava by building and analyzing a data flow graph and automatically selecting, configuring and inserting Vivado HLS directives and simple code transformations. This approach contrasts to the existing approaches by proposing a more generic framework that supports a larger variety of inputs, rather than the specific or limited use cases that are more often found in those approaches. The proposed framework has two stages: the first stage builds and simplifies a DFG obtained from the input source code, while the second stage applies code optimization strategies. These strategies are guided by simple heuristics that analyze the DFG in order to reach a decision. It is also possible to use the framework as a validation and automation tool for an existing code restructuring tool. The framework was evaluated by optimizing a wide range of input applications, synthesizing the optimized versions and comparing their latency and resource usage to their respective unoptimized versions. Results show that the framework can achieve significant latency speedups in all tested examples, which can range from 3×to 58×, depending on the benchmark. Resource usage is also kept under the FPGA limits in most cases, except when a loop with a complex control flow or a high number of iterations is completely unrolled. An user-configured strategy for simple loops can also achieve better results for those cases than the ones achieved by the generic strategies. Finally, code optimized automatically by the framework was also compared to code optimized manually by an expert, and the results are similar in most cases. This, coupled with the advantage of being a near-instantaneous way of optimizing code without requiring the time and effort of an experienced developer, shows that the proposed framework is a competitive alternative to manual code optimization, and therefore fulfills the main objective of this dissertation.

6.2 Future Work

The future improvements to the proposed framework are threefold. Firstly, there is space for the current heuristics to be improved. The most logical step would be to update the loop unrolling heuristic in order to only do partial unrolling of loops with a high number of iterations or with a complex control flow on their bodies, or to limit the number of resources used when fully unrolling a loop. This would address the main limitation identified in the framework. The array partitioning heuristic could also be improved in order to come up with custom partitioning factors for larger arrays. As for loop pipelining, the framework could try to estimate the required II, so that it could relax the II constraint in order to lead to a less resource-demanding design. This could also be provided as an option to the user, who could decide whether to use the estimated II, or keep the heuristic with no specified II and let Vivado HLS achieve the lowest possible value at the expense of a higher resource usage.

The second set of improvements are related to including new strategies using the other Vivado HLS directives, as described in Section 4.1.6. These would need to be prioritized, and the most relevant ones would require heuristics, such as loop merging and loop flattening. The framework can also support code restructuring performed directly over the AST rather than through heuristics, so transformations of that kind could also be explored. Another strategy would be to use the framework in conjunction with the existing trace-based approach. While the framework can currently instrument code for it, its usage is independent from the proposed directive-based approach, and a hybrid one could be a worthwhile investment.

Finally, the third set of improvements are related to expanding the C features supported by the framework. Currently, structs, custom data types and pointers to static memory are not supported, and adding support for those would expand the type of source code that could be analyzed and restructured. This would also allow to evaluate the framework with more complex benchmarks.

Appendix A

Framework API

Clava exposes the framework as a LARA library named clava.hls. This library contains an API comprised of static methods split across three sub-packages. They are the following:

- clava.hls.HLSAnalysis
  - `optimizeWithDirectives(funNames, loadStores)` - optimizes the provided functions using the proposed directive-based approach.
    - `funNames` - a list with the name of the functions to optimize
    - `loadStores` - the load/stores parameter (positive integer). This is an optional value, and if it is not specified, the load/stores strategy is not applied
  - `canBeInstrumented(funNames)` - checks if the provided functions can be instrumented and passed on to the trace-based approach. The result, for each function, is provided on the console output
    - `funNames` - a list with the name of the functions to validate
- clava.hls.MathAnalysis
  - `mathCompare()` - outputs to the console each call to a math.h function found in the current program, the argument types of each function and the type of the arguments being passed on each call
  - `mathReport(csv, name)` - counts the number of occurrences of each math.h function on the current program, and outputs them to a CSV file
    - `csv` - the name of the CSV file. If the name provided corresponds to an existing file, the results are appended; if not, then a new CSV file with the provided name is created
    - `name` - the name of the program being analyzed. If omitted, the program name defaults to the name of its folder
  - `mathReplace()` - replaces all calls to mathematical functions with float arguments by the float versions of those functions, while also applying the code transformations described in Section 4.3

- clava.hls.TraceInstrumentation
  - `instrument(funName)` - instruments the function with the provided name
    - `funName` - the name of the function to instrument

Listing A.2 provides an example of the output produced by the LARA aspect in Listing A.1. This aspect is applied to the DCT benchmark, and it passes "dct" as the name of the function to optimize. The output code itself is present in Listing A.3. An alternative way to interact with library would be to first select the function itself using LARA, but that is not necessary since the library can recognize that a string with the name is being passed and it performs the selection step internally.

```
1 import clava.hls.HLSAnalysis;
2
3 aspectdef Example
4   HLSAnalysis.optimizeWithDirectives("dct");
5   HLSAnalysis.canBeInstrumented("dct");
6 end
```

Listing A.1: Usage example for the HLSAnalysis library

```
1  HLS: Optimizing with directives
2  HLS: starting HLS restructuring
3  HLS: file "dct.dot" saved to "_HLS_graphs\dct.dot"
4  HLS: reporting the cost of each subgraph
5  HLS: file "features_dct.csv" saved to "_HLS_reports\features_dct.csv"
6  HLS: detecting if arrays can be turned into streams
7  HLS: detecting if function calls can be inlined
8  HLS: defining unrolling factor for nested loops
9  HLS: found master loop "loop i" (trip count = 8)
10 HLS: found master loop "loop i" (trip count = 8)
11 HLS: trying to unroll loop nest starting by bottom loop "loop k" (trip count =
       8)
12 HLS: trying to unroll loop nest starting by bottom loop "loop k" (trip count =
       8)
13 HLS: unrolling 2 loops
14 HLS: unrolling "loop k" (trip count = 8) fully
15 HLS: unrolling "loop k" (trip count = 8) fully
16 HLS: detecting if code regions can be pipelined
17 HLS: pipelining body of loop "loop j" with  undetermined II
18 HLS: pipelining body of loop "loop j" with  undetermined II
19 HLS: finished HLS restructuring
20 HLS:
       ------------------------------------------------------------------------
21 HLS: checking if function can be turned into a trace
22 HLS: function can be turned into a trace!
```

Listing A.2: Information outputted by Clava when running the framework for the DCT benchmark

```
1  int const CosBlock[8][8] = {88, 122, /*abridged*/ 47, -24};
2
3  void dct(int InIm[8][8], int TempBlock[8][8], int CosTrans[8][8], int OutIm
       [8][8]) {
4      #pragma HLS array_partition variable=InIm complete
5      #pragma HLS array_partition variable=TempBlock complete
6      #pragma HLS array_partition variable=CosTrans complete
7      #pragma HLS array_partition variable=OutIm complete
8      int i;
9      int j;
10     int k;
11     int aux;
12     for(i = 0; i < 8; i++) for(j = 0; j < 8; j++) {
13         #pragma HLS pipeline
14         aux = 0;
15         for(k = 0; k < 8; k++) {
16             #pragma HLS unroll
17             aux += InIm[i][k] * CosTrans[k][j];
18         }
19         TempBlock[i][j] = aux;
20     }
21     for(i = 0; i < 8; i++) for(j = 0; j < 8; j++) {
22         #pragma HLS pipeline
23         aux = 0;
24         for(k = 0; k < 8; k++) {
25             #pragma HLS unroll
26             aux += TempBlock[k][j] * CosBlock[i][k];
27         }
28         OutIm[i][j] = aux;
29     }
30 }
```

Listing A.3: Code generated by Clava after running the framework for the DCT benchmark (gray regions identify the directives inserted automatically)

Appendix B

Benchmark Code

This Appendix presents the code of all the benchmarks used for the evaluation. Some examples, such as DCT and the SVM, were already shown in other sections of the dissertation, but are repeated here for completeness.

B.1 Machine Learning Benchmarks

```
1  //Macro to choose the benchmark parameters
2  #define PROFILE_1
3
4  #ifdef PROFILE_1
5  #define NUM_FEATURES 32       //5 // number of features used
6  #define K 3                   // value of k in kNN
7  #define NUM_CLASSES 8         // number of classes considered
8  #define NUM_KNOWN_POINTS 1000 //8 //instances of the model used after training
9  #define ftype float           // type used for features
10 #define dtype float           // type used for distance calculation
11 #define ctype char            // type used for class ID, an integer
12
13 #endif
14
15 #ifdef PROFILE_2
16 #define NUM_FEATURES 64       //5 // number of features used
17 #define K 3                   // value of k in kNN
18 #define NUM_CLASSES 8         // number of classes considered
19 #define NUM_KNOWN_POINTS 1000 //8 //instances of the model used after training
20 #define ftype float           // type used for features
21 #define dtype float           // type used for distance calculation
22 #define ctype char            // type used for class ID, an integer
23
24 #endif
25 #ifdef PROFILE_3
26 #define NUM_FEATURES 128      //5 // number of features used
```

```
27 #define K 3                    // value of k in kNN
28 #define NUM_CLASSES 8          // number of classes considered
29 #define NUM_KNOWN_POINTS 1000 //8 //instances of the model used after training
30 #define ftype float            // type used for features
31 #define dtype float            // type used for distance calculation
32 #define ctype char             // type used for class ID, an integer
33
34 #endif
35
36 #ifdef PROFILE_4
37 #define NUM_FEATURES 32        //5 // number of features used
38 #define K 3                    // value of k in kNN
39 #define NUM_CLASSES 8          // number of classes considered
40 #define NUM_KNOWN_POINTS 10000 //8 //instances of the model used after training
41 #define ftype float            // type used for features
42 #define dtype float            // type used for distance calculation
43 #define ctype char             // type used for class ID, an integer
44
45 #endif
46
47 #define MAX_VALUE_OF_FEATURE 1
48
49 #if dtype == double
50 #define MAXDISTANCE DBL_MAX
51 #else
52 #define MAXDISTANCE FLT_MAX
53 #endif
54
55 #define sqr(x) ((x) * (x))
56
57 ctype classify3NN(ctype BestPointsClasses[K], dtype BestPointsDistances[K])
58 {
59
60     ctype c1 = BestPointsClasses[0];
61     dtype d1 = BestPointsDistances[0];
62
63     ctype c2 = BestPointsClasses[1];
64     dtype d2 = BestPointsDistances[1];
65
66     ctype c3 = BestPointsClasses[2];
67     dtype d3 = BestPointsDistances[2];
68
69     ctype classID;
70     dtype mindist = d1;
71
72     classID = (mindist > d2) ? c2 : c1;
73     mindist = (mindist > d2) ? d2 : d1;
74
75     classID = (mindist > d3) ? c3 : classID;
```

```
76      mindist = (mindist > d3) ? d3 : mindist;
77
78      classID = (c2 == c3) ? c2 : classID;
79      classID = (c1 == c3) ? c1 : classID;
80      classID = (c1 == c2) ? c1 : classID;
81
82      return classID;
83  }
84
85  void initializeBest(ctype BestPointsClasses[K], dtype BestPointsDistances[K])
86  {
87      for (int i = 0; i < K; i++)
88      {
89          BestPointsDistances[i] = MAXDISTANCE;
90          BestPointsClasses[i] = NUM_CLASSES;
91      }
92  }
93
94  void updateBest(dtype distance, ctype classifID, dtype BestPointsDistances[K],
95                  ctype BestPointsClasses[K])
96  {
97      dtype max = 0;
98      int index = 0;
99
100     for (int i = 0; i < K; i++)
101     {
102         dtype dbest = BestPointsDistances[i];
103         dtype max_tmp = max;
104         max = (dbest > max_tmp) ? dbest : max;
105         index = (dbest > max_tmp) ? i : index;
106     }
107
108     dtype dbest = BestPointsDistances[index];
109     ctype cbest = BestPointsClasses[index];
110
111     BestPointsDistances[index] = (distance < max) ? distance : dbest;
112     BestPointsClasses[index] = (distance < max) ? classifID : cbest;
113 }
114
115 ctype knn(ftype xFeatures[NUM_FEATURES], ftype knownFeatures[NUM_KNOWN_POINTS][
        NUM_FEATURES],
116           ctype knownClasses[NUM_KNOWN_POINTS])
117 {
118
119     dtype BestPointsDistances[K];
120     ctype BestPointsClasses[K];
121
122     initializeBest(BestPointsClasses, BestPointsDistances);
123
```

```
124     for (int i = 0; i < NUM_KNOWN_POINTS; i++)
125     {
126         dtype distance = (dtype)0;
127 
128         for (int j = 0; j < NUM_FEATURES; j++)
129         {
130             distance += (dtype)xFeatures[j] - (dtype)knownFeatures[i][j];
131         }
132         distance = sqrt(distance);
133         updateBest(distance, knownClasses[i], BestPointsDistances,
                BestPointsClasses);
134     }
135 
136     int classifyID = classify3NN(BestPointsClasses, BestPointsDistances);
137 
138     return classifyID;
139 }
```

Listing B.1: k-Nearest Neighbours (kNN) prediction function (in-house)

```
1  #define GAMMA 8
2  #define B 0
3  #define N_FEATURES 18
4  #define N_SUP_VECT 1248
5 
6  int svm_predict(float test_vector[N_FEATURES], float sup_vectors[N_FEATURES][
       N_SUP_VECT], float sv_coeff[N_SUP_VECT])
7  {
8      float diff;
9      float norma;
10     int sum = 0;
11     for (int i = 0; i < N_SUP_VECT; i++)
12     {
13         for (int j = 0; j < N_FEATURES; j++)
14         {
15             diff = test_vector[j] - sup_vectors[j][i];
16             diff = diff * diff;
17             norma = norma + diff;
18         }
19         sum = sum + (exp(-GAMMA * norma) * sv_coeff[i]);
20         norma = 0;
21     }
22     sum = sum - B;
23     return sum;
24 }
```

Listing B.2: Support Vector Machine (SVM) prediction function (Source: [25])

B.2 Matrix Manipulation Benchmarks

```
#define N 8

const int CosBlock[8][8] = {88, 122, 115, 103, 88, 69, 47, 24,
                            88, 103, 47, -24, -88, -122, -115, -69,
                            88, 69, -47, -122, -88, 24, 115, 103,
                            88, 24, -115, -69, 88, 103, -47, -122,
                            88, -24, -115, 69, 88, -103, -47, 122,
                            88, -69, -47, 122, -88, -24, 115, -103,
                            88, -103, 47, 24, -88, 122, -115, 69,
                            88, -122, 115, -103, 88, -69, 47, -24};

void dct(
    int InIm[N][N], int TempBlock[N][N],
    int CosTrans[N][N], int OutIm[N][N])
{
    int aux;

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            aux = 0;
            for (int k = 0; k < N; k++)
                aux += InIm[i][k] * CosTrans[k][j];
            TempBlock[i][j] = aux;
        }

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            aux = 0;
            for (int k = 0; k < N; k++)
                aux += TempBlock[k][j] * CosBlock[i][k];
            OutIm[i][j] = aux;
        }
}
```

Listing B.3: Discrete Cosine Transform using matrices (in-house)

```
#define NR1 8
#define NC1 4
#define NC2 8

void DSPF_sp_mat_mul_cplx_cn(float x1[2 * NR1 * NC1], float x2[2 * NC1 * NC2],
    float y[2 * NR1 * NC2])
```

```
6  {
7      float real;
8      float imag;
9      int i;
10     int j;
11     int k;
12
13     for (i = 0; i < NR1; i++)
14     {
15         for (j = 0; j < NC2; j++)
16         {
17             real = 0;
18             imag = 0;
19
20             for (k = 0; k < NC1; k++)
21             {
22                 real += (x1[i * 2 * NC1 + 2 * k] * x2[k * 2 * NC2 + 2 * j]
23                     - x1[i * 2 * NC1 + 2 * k + 1] * x2[k * 2 * NC2 + 2 * j +
                           1]);
24                 imag += (x1[i * 2 * NC1 + 2 * k] * x2[k * 2 * NC2 + 2 * j + 1]
25                     + x1[i * 2 * NC1 + 2 * k + 1] * x2[k * 2 * NC2 + 2 * j]);
26             }
27             y[i * 2 * NC2 + 2 * j] = real;
28             y[i * 2 * NC2 + 2 * j + 1] = imag;
29         }
30     }
31 }
```

Listing B.4: Complex matrix multiplication (source: [29])

```
1  #define A_ROW 10
2  #define A_COL 10
3  #define B_ROW 10
4  #define B_COL 10
5
6  void mult(float a_matrix[A_ROW][A_COL], float b_matrix[B_ROW][B_COL],
7            float c_matrix[A_ROW][B_COL])
8  {
9      float sum;
10
11     for (int i = 0; i < A_ROW; i++)
12     {
13         for (int j = 0; j < B_COL; j++)
14         {
15             sum = 0.0;
16             for (int k = 0; k < B_ROW; ++k)
17             {
18                 sum += a_matrix[i][k] * b_matrix[k][j];
```

```
            }
            c_matrix[i][j] = sum;
        }
    }
}
```

Listing B.5: Matrix multiplication (source: [30])

B.3 Filter-based Benchmarks

```
#define WIDTH_SIZE 800
#define HEIGHT_SIZE 600

const short K[] = {1, 2, 1, 2, 4, 2, 1, 2, 1};

void fir2D(unsigned char in[HEIGHT_SIZE * WIDTH_SIZE], unsigned char out[
    HEIGHT_SIZE * WIDTH_SIZE])
{
    for (int row = 0; row < HEIGHT_SIZE - 3 + 1; row++)
    {
        for (int col = 0; col < WIDTH_SIZE - 3 + 1; col++)
        {
            unsigned short sumval = 0;
            for (int wrow = 0; wrow < 3; wrow++)
            {
                for (int wcol = 0; wcol < 3; wcol++)
                {
                    sumval += (unsigned short)in[(row + wrow) * WIDTH_SIZE + (
                        col + wcol)]
                    * (unsigned short)K[wrow * 3 + wcol];
                }
            }
            sumval = sumval / 16;
            out[row * WIDTH_SIZE + col] = (unsigned char)sumval;
        }
    }
}
```

Listing B.6: Fir 2D filter (in-house)

```
void kalman(int Y[16], int A[16 * 16], int K[16 * 16], int G[16 * 16], int V
    [4])
{
    int i;
    int j;
```

```
5     int index;
6     int temp;
7     int X[16];
8
9     /* Initializing state Vector X */
10    for (i = 0; i < 16; i++)
11    {
12        X[i] = 0;
13    }
14
15    /* Initializing state Vector Y */
16    for (i = 12; i < 16; i++)
17    {
18        Y[i] = 0;
19    }
20
21    /* -- Computing state Vector X */
22    for (i = 0; i < 16; i++)
23    {
24        temp = 0;
25        for (j = 0; j < 16; j++)
26        {
27            index = i * 16 + j;
28            temp += (A[index] * X[j] + K[index] * Y[j]);
29        }
30        X[i] = temp;
31    }
32
33    /* -- Computing output Vector V */
34    // it only uses 4x16 elements of G
35    for (i = 0; i < 4; i++)
36    {
37        temp = 0;
38        for (j = 0; j < 16; j++)
39        {
40            index = i * 16 + j;
41            temp += (G[index] * X[j]);
42        }
43        V[i] = (temp * Y[i + 1]);
44    }
45 }
```

Listing B.7: Kalman filter (in-house)

```
1  #define N 100
2
3  void fir_k_a(int t[N], int o[N], int a[N], int m[N][N])
4  {
```

```
5     int i;
6     int j;
7
8     for (i = 0; i < N; i++)
9     {
10        t[i] = 0;
11        for (j = 0; j < 3; j++)
12            t[i] = t[i] + (a[i - j + 2] >> 2);
13    }
14    for (i = 0; i < N; i++)
15    {
16        o[i] = 0;
17        for (j = 0; j < N; j++)
18            o[i] = o[i] + m[i][j] * t[j];
19    }
20 }
```

Listing B.8: fir_K_A (source: [51])

B.4 Digital Signal Processing Benchmarks

```
1  #define M 160
2  #define N 10
3
4  int DSP_autocor(short ac[M], short sd[N + M])
5  {
6      int i;
7      int k;
8      int sum;
9
10     for (i = 0; i < M; i++)
11     {
12         sum = 0;
13         for (k = 0; k < N; k++)
14         {
15             sum += sd[k + M] * sd[k + M - i];
16         }
17         ac[i] = (sum >> 15);
18     }
19 }
```

Listing B.9: Autocorrelation (source: [29])

```
1  #define N 100
2
```

```
3  int DSP_dotprod(short x[N], short y[N])
4  {
5      int sum = 0;
6
7      for (int i = 0; i < N; i++)
8          sum += x[i] * y[i];
9
10     return sum;
11 }
```

Listing B.10: Dot Product (source: [29])

```
1  #define N 200
2
3  void gouraud(unsigned int rd, unsigned int r,
4               unsigned int gd, unsigned int g, unsigned int bd,
5               int b, int p[N])
6  {
7      unsigned int mask = 0xF800F800;
8      int i;
9      for (i = 0; i < N; i++)
10     {
11         r += rd;
12         g += gd;
13         b += bd;
14         p[i] = (r & mask) + ((g & mask) >> 5) + ((b & mask) >> 10);
15     }
16 }
```

Listing B.11: Gouraud shading (source: [29])

```
1  #define NPOINTS 64
2  #define ORDER 32
3
4  void latnrm(float data[NPOINTS], float outa[NPOINTS], float coefficient[NPOINTS
       ],
5              float internal_state[NPOINTS])
6  {
7      int i;
8      int j;
9
10     float left;
11     float right;
12     float top;
13     float bottom = 0;
14     float sum; /*  */
```

```
15      for (i = 0; i < NPOINTS; i++)
16      {
17          top = data[i];
18          for (j = 1; j < ORDER; j++)
19          {
20              left = top;
21              right = internal_state[j];
22              internal_state[j] = bottom;
23              top = coefficient[j - 1] * left - coefficient[j] * right;
24              bottom = coefficient[j - 1] * right + coefficient[j] * left;
25          }
26          internal_state[ORDER] = bottom;
27          internal_state[ORDER + 1] = top;
28          sum = 0.0;
29          for (j = 0; j < ORDER; j++)
30          {
31              sum += internal_state[j] * coefficient[j + ORDER];
32          }
33          outa[i] = sum;
34      }
35  }
```

Listing B.12: Lattice Filter (source: [30])

B.5 Load/Stores Evaluation

```
1   #define N 2000
2
3   void computeGradient(float grad[N], float feature[N], float scale)
4   {
5       for (int i = 0; i < N; i++)
6           grad[i] = scale * feature[i];
7   }
```

Listing B.13: computeGradient function (source: [60])

```
1   #define N 2000
2
3   void updateParameter(float param[N], float grad[N], float scale)
4   {
5       for (int i = 0; i < N; i++)
6           param[i] += scale * grad[i];
7   }
```

Listing B.14: updateParameter function (source: [60])

B.6 Comparison to Manual Code Restructuring

```
#define Ns 32
#define Ny 64
#define Nz 512
#define Nm 1024

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 B.15: Filter subband (source: [31])

References

[1] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, pp. 332–354, Mar 2015.

[2] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," *Advances in Intelligent Systems and Computing Advances in Systems Science*, p. 483–492, 2014.

[3] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D.-J. Lee, "Real-time optical flow calculations on fpga and gpu architectures: A comparison study," *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, 2008.

[4] S. Singh, "Computing without processors," *Queue*, vol. 9, p. 50, Jan 2011.

[5] "Intel fpga acceleration cards." Available at https://www.intel.com/content/www/us/en/programmable/solutions/acceleration-hub/platforms.html, Accessed last time in June 2020.

[6] Xilinx, "Alveo." Available at https://www.xilinx.com/products/boards-and-kits/alveo.html, Accessed last time in June 2020.

[7] Xilinx, "Zynq-7000 soc data sheet: Overview, ds190 (v1.11.1)." Available at https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, Accessed last time in February 2020, 2018.

[8] Avnet, "Zedboard." Available at http://zedboard.org/sites/default/files/product_briefs/5066-PB-AES-Z7EV-7Z020-G-V3c%20%281%29_0.pdf, Accessed last time in February 2020, 2018.

[9] "Amazon ec2 f1 instances." Available at https://aws.amazon.com/ec2/instance-types/f1/, Accessed last time in June 2020.

[10] "What are fpga - how to deploy." Available at https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service, Accessed last time in June 2020.

[11] "Ieee standard vhdl language reference manual," *IEEE Std 1076-1987*, pp. 1–218, 1988.

[12] "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, 2006.

[13] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, p. 56, Jan 2013.

[14] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *Design Test of Computers, IEEE*, vol. 26, pp. 8 – 17, 09 2009.

[15] SPeCS, "specs-feup/clava: C/c++ source-to-source tool based on clang." Available at http://specs.fe.up.pt/tools/clava/, Accessed last time in February 2020.

[16] A. C. Ferreira and J. M. P. Cardoso, "Unfolding and folding: a new approach for code restructuring targeting hls for fpgas," in *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, pp. 1–10, 2018.

[17] Xilinx, "Introduction to fpga design with vivado high-level synthesis, ug998 (v1.1)." Available at https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf, Accessed last time in February 2020, 2019.

[18] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup," *ACM Transactions on Embedded Computing Systems*, vol. 13, p. 1–27, Jan 2013.

[19] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, and F. e. a. Ferrandi, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, p. 1591–1604, 2016.

[20] "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[21] "Bambu." Available at https://panda.dei.polimi.it/?page_id=31, Accessed last time in May 2020.

[22] R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler," *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[23] Xilinx, "Vivado hls optimization methodology guide, ug1270 (v2018.1)." Available at https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf, Accessed last time in February 2020, 2018.

[24] "High-level synthesis for any fpga | legup computing." Available at https://www.legupcomputing.com/, Accessed last time in June 2020.

[25] V. Tsoutsouras, K. Koliogeorgi, S. Xydis, and D. 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*, vol. 88, p. 127–147, Jul 2017.

[26] C. Li, Y. Bi, Y. Benezeth, D. Ginhac, and F. Yang, "High-level synthesis for fpgas: code optimization strategies for real-time image processing," *Journal of Real-Time Image Processing*, vol. 14, p. 701–712, Apr 2017.

[27] J. Huang and T. Leng, "Generalized loop-unrolling: a method for program speedup," *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET99 (Cat. No.PR00122)*.

[28] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, (New York, NY, USA), p. 9–18, Association for Computing Machinery, 2013.

[29] T. Instruments, "Tms320c6000 dsp library (dsplib)." Available at http://www.ti.com/tool/SPRC265, Accessed last time in February 2020.

[30] "Utdsp benchmark suite." Available at https://www.eecg.utoronto.ca/~corinna/DSP/infrastructure/UTDSP.html, Accessed last time in May 2020.

[31] J. M. P. Cardoso, P. C. Diniz, Z. Petrov, K. Bertels, M. Hübner, H. V. Someren, F. Gonçalves, J. G. F. D. Coutinho, G. A. Constantinides, and B. e. a. Olivier, "Reflect: Rendering fpgas to multi-core embedded computing," *Reconfigurable Computing*, p. 261–289, 2011.

[32] S. Cheng and J. Wawrzynek, "High level synthesis with a dataflow architectural template," in *Proceedings of the Second International Workshop on Overlay Architectures for FPGAs (OLAF 2016)*, pp. 14–19, Mar 2016.

[33] M. A. Özkan, A. Pérard-Gayot, R. Membarth, P. Slusallek, R. Leissa, S. Hack, J. Teich, and F. Hannig, "Anyhls: High-level synthesis with partial evaluation," 2020.

[34] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "Anydsl: a partial evaluation framework for programming high-performance libraries," *Proceedings of the ACM on Programming Languages*, vol. 2, p. 1–30, Nov 2018.

[35] "Intel fpga sdk for opencl software technology." Available at https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, Accessed last time in June 2020.

[36] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," *ACM Trans. Archit. Code Optim.*, vol. 14, Aug. 2017.

[37] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Korner, and W. Eckert, "Hipacc: A domain-specific language and compiler for image processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, p. 210–224, Jan 2016.

[38] X. Gao, J. Wickerson, and G. A. Constantinides, "Automatically optimizing the latency, area, and accuracy of c programs for high-level synthesis," *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA 16*, Feb 2016.

[39] J. Dongarra and P. Luszczek, *Livermore Loops*, pp. 1041–1043. Boston, MA: Springer US, 2011.

[40] L.-N. Pouchet, "Polybench/c - the polyhedral benchmark suite." Available at https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, Accessed last time in June 2020.

[41] J. M. P. Cardoso, J. G. F. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, and F. Gonçalves, "Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach," *Software: Practice and Experience*, vol. 46, p. 251–287, Nov 2014.

[42] 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," *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012.

[43] "Special-purpose computing systems, languages and tools (specs)." Available at http://specs.fe.up.pt/, Accessed last time in February 2020.

[44] "Clang: a c language family frontend for llvm." Available at https://clang.llvm.org/, Accessed last time in June 2020.

[45] P. Pinto, T. Carvalho, J. Bispo, M. A. Ramalho, and J. M. Cardoso, "Aspect composition for multiple target languages using lara," *Computer Languages, Systems Structures*, vol. 53, p. 1–26, 2018.

[46] SPeCS, "Clava." Available at https://github.com/specs-feup/clava, Accessed last time in June 2020.

[47] S. S. Muchnick, *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[48] "The dot language." Available at https://graphviz.org/doc/info/lang.html, Accessed last time in June 2020.

[49] "gvpr(1) — graphviz — debian testing — debian manpages." Available at https://manpages.debian.org/testing/graphviz/gvpr.1.en.html, Accessed last time in June 2020.

[50] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, p. 1–19, July 1970.

[51] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[52] H. Arabnejad, J. Bispo, J. Barbosa, and J. Cardoso, "Autopar-clava: An automatic parallelization source-to-source tool for c code applications," pp. 13–19, 01 2018.

[53] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*, pp. 1192–1195, May 2008.

[54] "Snu real-time benchmarks." Available at http://www.cprover.org/goto-cc/examples/snu.html, Accessed last time in June 2020.

[55] L. Kalms, A. Podlubne, and D. Göhringer, "Hiflipvx: An open source high-level synthesis fpga library for image processing," *Lecture Notes in Computer Science Applied Reconfigurable Computing*, p. 149–164, 2019.

[56] R. Griffith and F. Pong, "Microblaze system performance tuning," Aug 2008.

[57] "Mingw | minimalistic gnu for windows." Available at http://www.mingw.org/, Accessed last time in June 2020.

[58] id Software, "Quake iii arena." Available at https://github.com/id-Software/Quake-III-Arena, Accessed last time in July 2020.

[59] "Xilinx." Available at https://www.xilinx.com/, Accessed last time in July 2020.

[60] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.