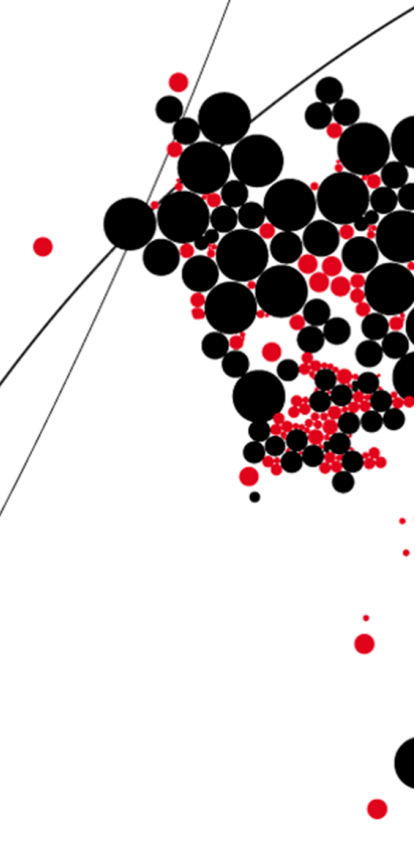
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Automatically map an algorithmic description to reconfigurable hardware using the Decoupled Access-Execute architecture

Darrel Griët

M.Sc. Thesis
October 2021



Supervisors:
Dr. Ir. S.H. Gerez
Dr. Ir. N. Alachiotis
Dr. Ir. A.B.J. Kokkeler

Computer Architecture for Embedded Systems
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

Moore's law is proclaimed to be declining while the data science field processes more and more data. Traditionally, these algorithms were deployed on general purpose processors, but as data sets are growing so is the execution time of the algorithm. This has the potential to limit innovations in research and development. Recently, there is a trend where data scientists are exploring alternative solutions to accelerate their algorithms. One such alternative is the use of hardware accelerators on Field-Programmable Gate Arrays (FPGAs). However, an issue arises because it is not straightforward and it is time consuming to map the traditionally sequential algorithms to reconfigurable hardware. High-level synthesis (HLS) tools improve this by mapping sequential C/C++ specifications to an FPGA register transfer level description. This process is however not fully automatic, manual changes are still required. Furthermore, there is evidence that changing the structure of the code to the Decoupled Access-Execute (DAE) architecture increases the speedup of the algorithm as it improves the memory accessing part. The DAE architecture consists of separating the memory accessing patterns from the computational parts in the C/C++ code.

In this thesis a framework is proposed that automatically transforms the structure of an algorithm written in the C/C++ programming language to the DAE architecture. The use of the DAE architecture creates separation of concerns. As the memory accessing and memory address calculation logic is moved into dedicated units that operates independently of other units, the computational part has access to memory only via the dedicated memory accessing units.

The framework does not recognize all different types of memory accessing patterns, therefore it is evaluated against a subset of the algorithms provided by the MachSuite benchmark. The runtime of the algorithm is measured then it is transformed into the DAE architecture and the appropriate HLS directives are automatically added and again the runtime is measured. Depending on the benchmark a maximum speedup of 1.63x is observed while in the worst case a negligible speedup is observed, showing that the transformation highly depends on the algorithm. In addition to runtime measurement, power and area usage is also measured. Power usage appears to be directly linked to the speedup: The power usage is increased

for the algorithms where the speedup also is increased. The amount of area used for the transformed algorithm also increases for those.

Contents

List of acronyms

ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
BRAM	Block RAM
CDFG	Control Data Flow Graph
CFG	Control-Flow Graph
CGRA	Coarse-Grained Reconfigurable Array
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DAE	Decoupled Access-Execute
DFG	Data Flow Graph
DSP	Digital Signal Processing
FF	Flip-Flop
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPC	High-Performance Computing

IR Intermediate Representation

ISA Instruction Set Architecture

LUT Look-Up Table

PDG Program Dependence graph

PL Programmable Logic

PS Processor System

RTL Register Transfer Level

SoC System on a chip

VHDL VHSIC Hardware Description Language

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

State-of-the-art data science engineering such as bioinformatics and machine learning process large complex sets of data. Traditionally, these data sets are processed on conventional processors. As these data sets are growing in size and complexity a need for more powerful processors is growing. Now that Moore's law, an observation that depicts that the number of transistors in an integrated circuit doubles every two years, has been proclaimed to be nearing its end [3] and the growing need for faster processors increases, there is a visible shift towards more specialized hardware that is used to process these data sets. Instead of using a processor only, there is now a trend where dedicated accelerators are deployed alongside the processor. These dedicated accelerators have the capability to increase the performance of an algorithm by implementing it partly or entirely in the accelerator. Various technologies exist that allow for these accelerators to be implemented, varying from deeply integrated into hardware (Application-Specific Integrated Circuits (ASICs)) to more flexible platforms (Graphics Processing Unit (GPU) and reconfigurable hardware). This thesis specifically targets reconfigurable hardware as it offers a high flexibility of algorithm implementation onto the hardware while also allowing for it to be altered once implemented.

There exists multiple different types of reconfigurable hardware namely: Complex Programmable Logic Devices (CPLDs), Coarse-Grained Reconfigurable Arrays (CGRAs) and Field-Programmable Gate Arrays (FPGAs). The reconfigurable hardware that thesis will focus on is the industry dominating FPGA. This is a silicon chip that has the ability to be configured after it has been manufactured. FPGAs have grown in popularity due to their high flexibility at a relatively high efficiency. This flexibility includes the possibility to reconfigure the hardware to allow for parallel computation. FPGAs are not only growing in interest for bioinformatics but also for other fields like High-Performance Computing (HPC) and machine learning. All these fields process a lot of data in complex algorithms. Specialized hardware accelerators for these algorithms improve the throughput and speedup.

Most algorithms are implemented using an imperative programming language, such as C/C++, on a processor. A hardware description language (HDL) is used for the implementation of the logic on an FPGA. Conceptually HDLs and imperative programming languages differ in that an imperative programming language describes how to realize an algorithm while a HDL describes the digital logic of an FPGA. This means that an algorithm written in imperative programming languages can not be used directly on an FPGA.

1.1 Problem definition and Research questions

Even though FPGAs have many advantages, it is considered hard, time consuming and an error prone task to map complex algorithms to FPGAs because the developer needs to know hardware details in order for the algorithm to be efficiently and fully utilized [4] [5].

Recently, high-level synthesis (HLS) tools are gaining interest as they attempt to mitigate these issues by allowing the engineer to use the familiar C/C++ specification to describe the hardware [4]. A HLS tool transforms this specification into a register transfer level (RTL) implementation that can be synthesized for an FPGA. This is beneficial as software and hardware developers can implement the C/C++ code that was initially written for traditional processors and now target FPGAs, taking advantage of the parallel architecture of FPGAs. This greatly reduces the time-to-market which makes FPGAs feasible to more software projects [5].

While HLS tools improve the main issues with regards to programming an FPGA this is not completely automated and it requires manual changes so that the architecture of the FPGA is efficiently utilized.

Additionally, there is evidence that changing the software architecture to the Decoupled Access-Execute (DAE) architecture prior to using HLS tools increases the speedup of algorithms by 1.89x [6] to 2x [2] due to the more efficient data transfers. This speedup is an average of a diverse set of applications, namely a general matrix multiplication (gemm), a breadth-first search (bfs), a sparse matrix/vector multiplication (spmv), molecular dynamics (md), a stencil computation, the Needleman-Wunsch algorithm and the Viterbi algorithm.

This thesis builds upon the observation that the use of the DAE architecture allows for the creation of C/C++ code that can be optimized for an HLS tool in a systematic, structured and general way. The DAE architecture splits the algorithm written in C/C++ to access and execute components, creating separation of concerns. This allows for specific optimizations that are relevant for accessing external memory and further exploration on optimizations possible on the computational execution parts. The use of this standard structure has a threefold benefit, (1) allows

for automating the process and (2) results in a more efficient hardware design while also (3) resulting in a potential speedup.

This thesis presents a framework that automatically translates C/C++ code to a C/C++ code that is optimized for use with a HLS tool by using the DAE paradigm. The generated C/C++ code from the framework is expected to be human readable such that the algorithm designer can still experiment by improving other parts of the algorithm. While also having the aforementioned benefits.

Following this the main research question is formed:

Which steps are required to automatically translate C/C++ code to efficient HLS code for FPGAs using the DAE paradigm?

The DAE paradigm splits the architecture of the C/C++ code to access and execute units, a direct research sub-question related is:

1. How can one extract memory and computational parts from the C/C++ code?

These different units need to be interconnected which introduces the following research sub-questions:

2. How can one solve dependencies (data access) within the different access and execute units?

3. How can one establish correct communication between the different access and execute units?

To evaluate the implementation against industry standard benchmarks (for example OpenDwarfs [7] or MachSuite [8]) and real-world algorithms. The following research sub-question is relevant:

4. How does the execution time compare against other hardware implementations (baseline benchmark, manually optimized)?

1.2 Contributions

Mapping an existing algorithm onto an FPGA is considered a hard, time consuming and error prone task. HLS tools attempt to solve these shortcomings by transforming a high-level description of an algorithm into a hardware description. While this solves some of these shortcomings there is still the need for manual changes as well as there has been an observation that changing the architecture of the algorithm prior to using it in the HLS tool improves the speedup.

This thesis presents a fully automated framework that translates C/C++ code to C/C++ code that is optimized for use with a HLS tool using the DAE architecture. This reduces the need for low-level knowledge of the targeted FPGA, lowering the learning curve for software and hardware developers to target FPGAs. The generated C/C++ code is human readable allowing for further manual optimizations to the algorithm by more experienced developers. The automated nature of the framework reduces the initial time required to get an algorithm efficiently targeted for FPGAs.

1.3 Report organization

Chapter 2 describes the background information for the relevant topics and shows what their limitations are.

The related work is described in Chapter 3, its range varies from compiler technologies to high-level synthesis tools and their approach as to how to synthesize to a register transfer level description.

The design of the framework is described in Chapter 4. How different tools and techniques are used to implement this framework is described in Chapter 5.

Following the implementation the framework is evaluated in Chapter 6. Lastly, the conclusions and recommendations are given in Chapter 7.

Chapter 2

Background

This chapter describes the background concepts that are relevant for this thesis. It is organized as follows: Section 2.1 gives a brief overview of some important aspects of an FPGA. Followed by the HLS concept and toolings in Section 2.2. This will then lead into the use of the DAE architecture in Section 2.3. Lastly, the topic of source-to-source translation is described in Section 2.4.

2.1 FPGA

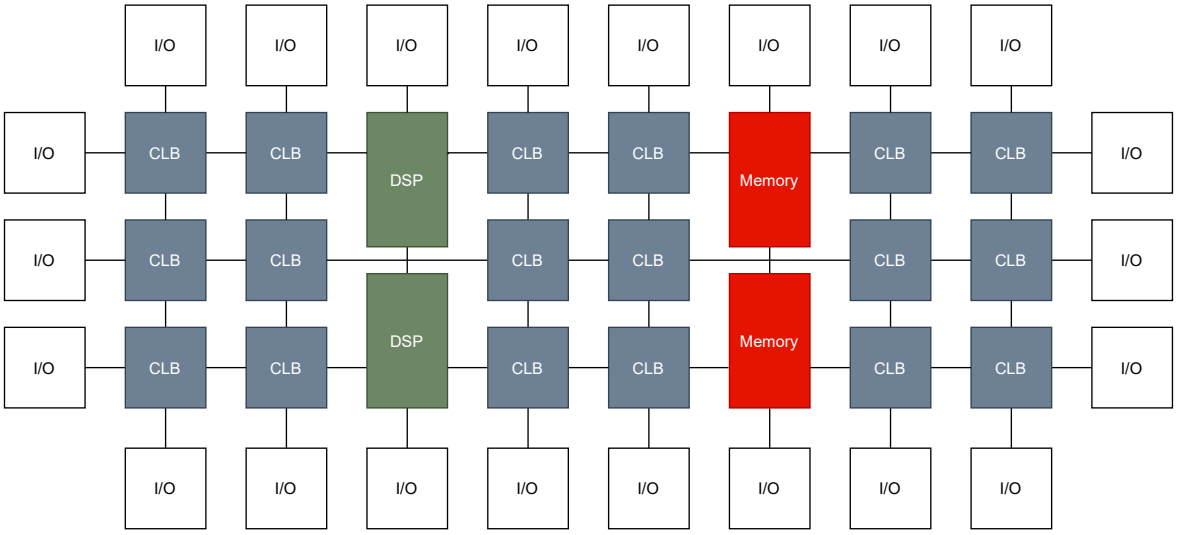


Figure 2.1: Simplified architecture of an FPGA

An FPGA is a silicon chip that can be configured after it has been manufactured. It consists of Configurable Logic Block (CLB) a programmable interconnect and input and output. The way these CLBs and interconnect is configured happens after the chip has been manufactured.

CLBs are the fundamental building blocks of an FPGA. A CLB consists of multiple Look-Up Tables (LUTs), memory and shift register logic, arithmetic functions and multiplexers which are grouped in a slice. Figure 2.1 shows a simplified architecture of an FPGA.

Nowadays, FPGAs contain additional specialized blocks such as multipliers and digital signal processing (DSP) blocks to increase computational density and efficiency. A recent trend is to also include a hard processor system, often an ARM processor core. This can run conventional software while having the ability to call custom hardware accelerators.

The CLBs and the programmable interconnect make FPGAs very powerful as it allows the engineer to design a digital hardware circuit after the chip has been produced. The engineer even has the ability to deploy the a different hardware circuit when the FPGA has already been shipped to its customers (in the field).

The design for an FPGA is written in a HDL. The most notable ones are VHSIC Hardware Description Language (VHDL) and Verilog.

2.2 High Level Synthesis

FPGAs are silicon chips that are configured after it was manufactured, its architecture is highly parallel and does not have a predefined Instruction Set Architecture (ISA), that is used for Central Processing Units (CPUs). HDLs are used to describe FPGAs as these represent a level at which digital logic can be described. A direct consequence is that more general purpose programming languages, like C/C++, can not be used.

HLS attempts to solve that by allowing the engineer to write a hardware description in a (often) C/C++ programming language (most commonly ANSI C [4]). The HLS tool transforms this into a hardware description (Often VHDL or Verilog) that can be synthesized onto an FPGA.

Figure 2.2 shows the design flow of a HLS tool. The engineer supplies the HLS tool with the algorithm (written in C/C++) and a test bench (also written in C/C++) to verify functional correctness. The most important part about HLS is scheduling, it determines when a statement in C/C++ is scheduled for execution, depending on constraints, multiple statements can be scheduled in parallel.

Traditionally, the RTL is verified using a test bench written in a HDL, with HLS tools it is not needed to write a test bench in this HDL. Instead, the supplied test bench written in C/C++ is also used to verify functional correctness of the RTL implementation. The test bench allows for verification of functional correctness for the algorithm written in C/C++ and also the synthesized RTL.

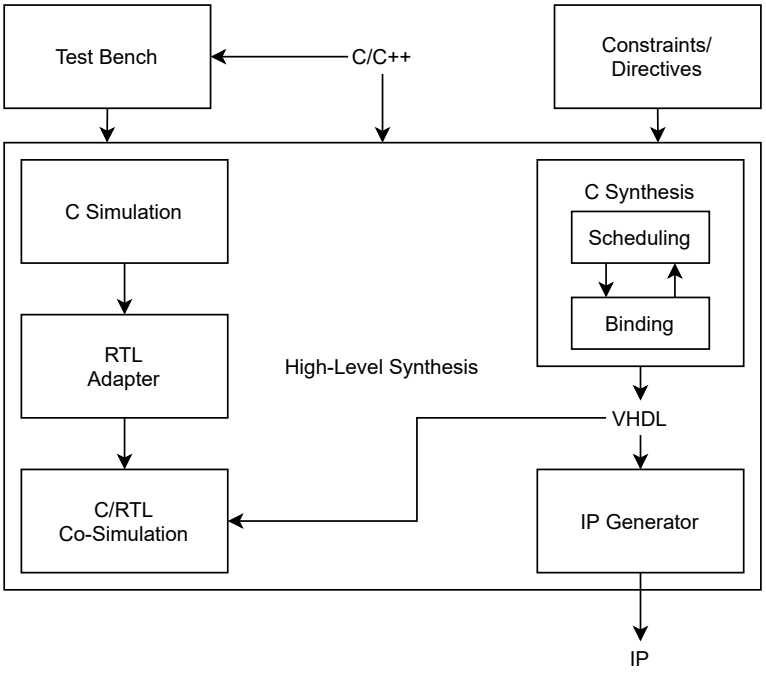


Figure 2.2: HLS design flow overview

The important aspect being that the algorithm is written in C/C++, this is then verified for functional correctness using C/C++ simulation. The synthesized RTL is also verified against the C/C++ simulation for functional correctness.

All, this reduces the high expertise needed for developing an algorithm for FPGAs. But even with this reduction, there is still the requirement to apply manual optimizations to the source code to make sure that the FPGA hardware is optimally used. An example for this is loop pipelining. Depending on the HLS tool used there are many more options to configure [9] depending on the architecture of the algorithm. Configuring these options wrongly can also result in degraded performance due to incorrect mapping.

2.3 Decoupled Access-Execute

The Decoupled Access-Execute architecture was originally designed for processors to improve performance [10]. It features a high degree of decoupling between access and execute operands. Separate program streams are responsible for either memory data accessing or computational execution. The computational stream, execute unit, never interacts with memory, it receives and stores its data via queues that are connected to the memory accessing streams, also known as an access unit.

Figure 2.3 gives an overview of the DAE architecture that will be used in this

thesis. All units must run in parallel, otherwise a unit will wait for data from units that have yet to be started, causing a deadlock. The fact that these units need to run in parallel is a benefit for FPGAs as they excel at running multiple tasks in parallel.

The DAE architecture shown in Figure 2.3 has a clear separation between the computational and the memory accessing parts of the algorithm. Essentially, the overall structure of the C/C++ code architecture remains the same when moving towards the DAE architecture. The main change is that loops are duplicated among the different units. For example, when the algorithm reads from memory x times then when moving to the DAE architecture the same number of reads are to be expected otherwise the units can get out of synchronization.

The DAE architecture proposed by Smith [10] uses two units: an access unit and an execute unit. Each have their own program stream and their own dedicated processor. Blocking queues are used to ensure that the processors stay in synchronization. In this thesis a single execute unit will be used, this represents the implementation of the algorithm as it was provided by the engineer. Depending on the number of memory accesses multiple access units will be used.

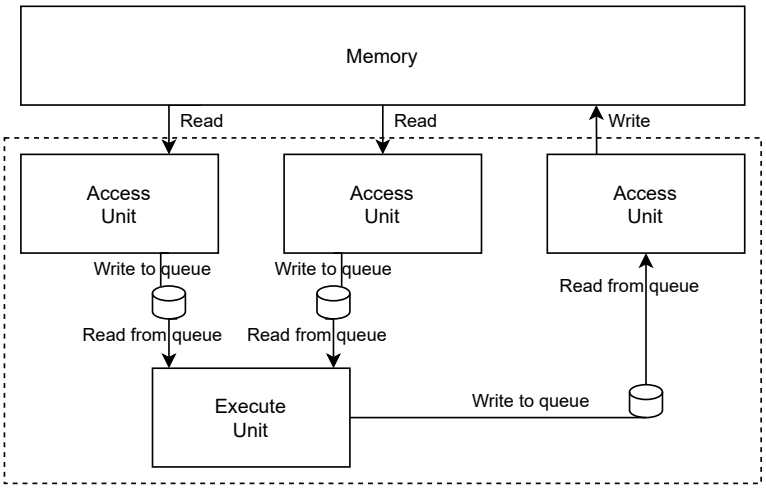


Figure 2.3: The Decoupled Access-Execute architecture

2.4 Source-to-source translation

Source-to-source translation works on a high-level programming language and translates that into another high-level programming language. Source-to-source translation is a strategy that is often used for code refactoring.

There exist the ROSE compiler framework [11] that allows for source-to-source transformations, but it lacks the capability to apply a wide range of code transformation, for this reason it is not widely used in the compiler and HPC community.

Instead, the LLVM compiler infrastructure is gaining traction, due to its modular design. A source-to-source translator described by Balogh et al. [12] has moved away from the ROSE compiler framework in favour of the LLVM libTooling as this supposedly gives a wider range of code transformations.

While source-to-source translation happens at the high-level programming level. Compilers have the task to translate a high-level programming language to another low-level programming language. A compiler generally works on three different stages: front end, intermediate representation (IR), back end. The front end is responsible for taking the high-level programming language and translating that to the intermediate representation. A lexer is used to create a list of tokens that represent the input code. The preprocessor has the ability fo manipulate the tokens, after which the tokens are parsed into a parse tree. The parser tree is transformed to an Abstract Syntax Tree (AST). The parser tree contains more information when compared to an AST. Finally, the AST is transformed into an IR.

The IR is optimized to improve performance and quality of the low-level programming language. At this stage a Control-Flow Graph (CFG) is built from the IR, which is used for static analysis of the IR. Compilers generate CFGs for the optimization of the IR. From CFG it is also possible to generate Data Flow Graph (DFG). It is also possible to generate a Control Data Flow Graph (CDFG) or Program Dependence graph (PDG) from the CFG. DFGs show graphically how data flows though an application. A node consists of a data transformation, while an edge indicates the flow of data. Namely the data dependencies become visible in this way.

```
1 int main() {
2     int v1;
3     int v2 = 0;
4     for (v1 = 0; v1 < 20; v1++) {
5         v2 += v1;
6     }
7     return v2;
8 }
```

Listing 2.1: The example input code

Listing 2.4 shows the AST generated by the code example shown in Listing 2.1. From top to bottom it shows a tree structure where a node has children nodes nested within. The individual nodes also define the meaning of the child nodes, for example: The for-loop has multiple children, but only the last node (*CompoundStmt*) contains information about the body of the loop. All others are related to the parameters of the loop (initialization, test expression, update statement).

The AST is translated into the IR shown in Listing 2.2. While the AST is already an abstraction of the input code, the IR shows an even larger abstraction, loops are

```
FunctionDecl 0x55d5278f99e0 <llvm_demo.c:4:1, line:11:1> line:4:5 main 'int ()'
`-CompoundStmt 0x55d5278f9e00 <col:12, line:11:1>
  |-DeclStmt 0x55d5278f9b00 <line:5:5, col:11>
  | `-VarDecl 0x55d5278f9a98 <col:5, col:9> col:9 used v1 'int'
  |-DeclStmt 0x55d5278f9bb8 <line:6:5, col:15>
  | `-VarDecl 0x55d5278f9b30 <col:5, col:14> col:9 used v2 'int' cinit
  |   `-IntegerLiteral 0x55d5278f9b98 <col:14> 'int' 0
  |-ForStmt 0x55d5278f9d80 <line:7:5, line:9:5>
  | |-BinaryOperator 0x55d5278f9c10 <line:7:10, col:15> 'int' '='
  | | |-DeclRefExpr 0x55d5278f9bd0 <col:10> 'int' lvalue Var 0x55d5278f9a98 'v1' 'int'
  | | `-IntegerLiteral 0x55d5278f9bf0 <col:15> 'int' 0
  | |-<<<NULL>>>
  | |-BinaryOperator 0x55d5278f9c88 <col:18, col:23> 'int' '<'
  | | |-ImplicitCastExpr 0x55d5278f9c70 <col:18> 'int' <LValueToRValue>
  | | | `-DeclRefExpr 0x55d5278f9c30 <col:18> 'int' lvalue Var 0x55d5278f9a98 'v1' 'int'
  | | `-IntegerLiteral 0x55d5278f9c50 <col:23> 'int' 20
  | |-UnaryOperator 0x55d5278f9cc8 <col:27, col:29> 'int' postfix '++'
  | | `-DeclRefExpr 0x55d5278f9ca8 <col:27> 'int' lvalue Var 0x55d5278f9a98 'v1' 'int'
  | `-CompoundStmt 0x55d5278f9d68 <col:33, line:9:5>
  |   `-CompoundAssignOperator 0x55d5278f9d38 <line:8:9, col:15> 'int' '+=' ComputeLHSTy='int' ComputeResultTy='int'
  |     |-DeclRefExpr 0x55d5278f9ce0 <col:9> 'int' lvalue Var 0x55d5278f9b30 'v2' 'int'
  |     `-ImplicitCastExpr 0x55d5278f9d20 <col:15> 'int' <LValueToRValue>
  |       `-DeclRefExpr 0x55d5278f9d00 <col:15> 'int' lvalue Var 0x55d5278f9a98 'v1' 'int'
  `-ReturnStmt 0x55d5278f9df0 <line:10:5, col:12>
    `-ImplicitCastExpr 0x55d5278f9dd8 <col:12> 'int' <LValueToRValue>
      `-DeclRefExpr 0x55d5278f9db8 <col:12> 'int' lvalue Var 0x55d5278f9b30 'v2' 'int'
```

Figure 2.4: The AST generated from Listing 2.1

replaced with label jumps, similar to assembly code.

A useful tool is to use a CFG for further code analysis. Figure 2.5 shows the CFG generated from the IR shown in Figure 2.1. The relation between the different labels are more clearly visible compared to the IR.

Code block *%4* is responsible for checking if the variable *v1* is still within the valid guard. If this is true, then it jumps to the *%7* code block, otherwise it jumps to *%14* which loads the variable *%v2* and returns that as the result of the main function. The *%7* code block handles the body of the for-loop: Sum *v1* and *v2* and store into *v2*. The code block *%11* is responsible for incrementing the loop guard *v1*.

```
define dso_local i32 @main() #0 !dbg !9 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %3, align 4, !dbg !16
  store i32 0, i32* %2, align 4, !dbg !17
  br label %4, !dbg !19
4:                                                ; preds = %11, %0
  %5 = load i32, i32* %2, align 4, !dbg !20
  %6 = icmp slt i32 %5, 20, !dbg !22
  br i1 %6, label %7, label %14, !dbg !23
7:                                                ; preds = %4
  %8 = load i32, i32* %2, align 4, !dbg !24
  %9 = load i32, i32* %3, align 4, !dbg !26
  %10 = add nsw i32 %9, %8, !dbg !26
  store i32 %10, i32* %3, align 4, !dbg !26
  br label %11, !dbg !27
11:                                               ; preds = %7
  %12 = load i32, i32* %2, align 4, !dbg !28
  %13 = add nsw i32 %12, 1, !dbg !28
  store i32 %13, i32* %2, align 4, !dbg !28
  br label %4, !dbg !29, !llvm.loop !30
14:                                               ; preds = %4
  %15 = load i32, i32* %3, align 4, !dbg !33
  ret i32 %15, !dbg !34
}
```

Listing 2.2: The IR used by a compiler infrastructure

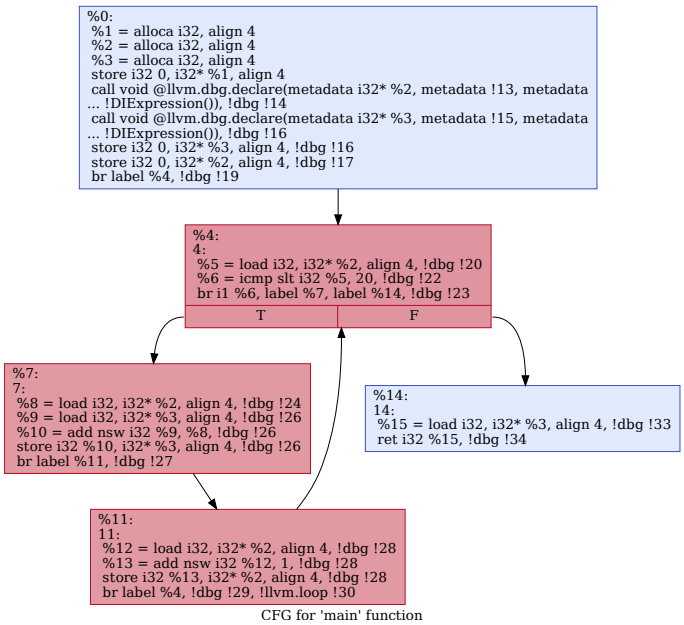


Figure 2.5: Control-Flow Graph

Chapter 3

Related work

This chapter will discuss related work and how it relates to this thesis. Section 3.1 gives an analysis into existing HLS tools and source-to-source tools. Section 3.2 will look at other related works that focus on the usage of the DAE architecture.

3.1 HLS and source-to-source translation

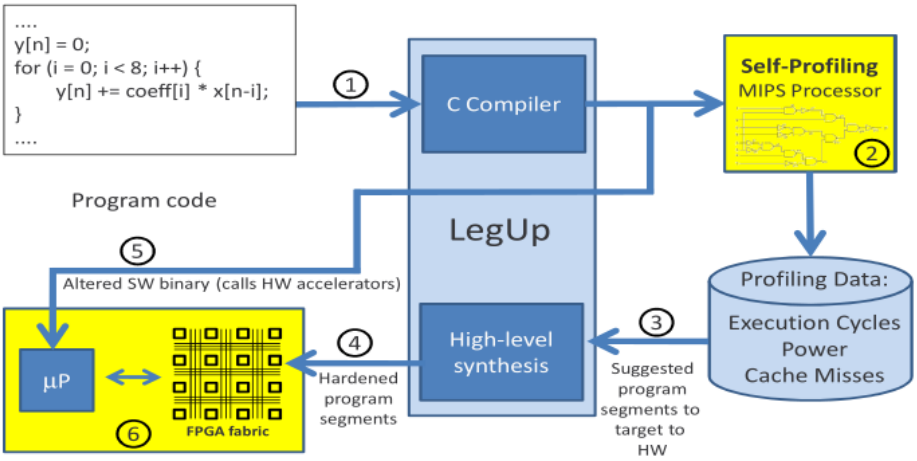


Figure 3.1: Design flow of the LegUp framework adapted from [1]

LegUp [1] is an open source HLS tool that uses the LLVM infrastructure to compile a standard C program to a hybrid architecture with a MIPS softcore processor and custom hardware accelerators. It specifically targets Intel FPGAs. The architecture shown in Figure 3.1 is such that it compiles C source code into a binary, this is executed on a MIPS processor. The MIPS processor is used to profile the binary. This way it can provide useful information on which sections of a program would benefit from a hardware implementation. The manually chosen sections should be appended to a file that is used by LegUp. LegUp then compiles these sections to

synthesizable Verilog. The Verilog is then synthesized to the FPGA implementation using the Altera/Intel FPGA vendor tool. Lastly the original C code is modified to call the custom hardware accelerators instead of the software implementation. LegUp utilizes the LLVM infrastructure by performing optimizations in the LLVM frontend passes. Then a LegUp code generator is used in the LLVM backend to create the Verilog output from the LLVM IR. LegUp also employs loop pipelining, but only loops where the loop body consists of a single basic block can be pipelined by LegUp. As such it is recommended to avoid if-else statements, replacing those by a C ternary operator (*condition* ? *expression* : *expression*). This means that the input source code needs manual changes to reduce the resources and improve pipelining.

The Merlin Compiler [13] is a closed source source-to-source compiler for FPGAs. It performs pre-synthesis source-to-source modifications. The Merlin Compiler can use a variety of vendor HLS tools such as Xilinx SDAccel and Altera OpenCL SDK, but they also provide their own HLS tool. The source-to-source compiler is implemented as multiple backend optimization passes in the LLVM compiler framework. The Merlin Compiler also verifies the output using CPU emulation. A runtime manager is responsible for scheduling tasks on the FPGA and, if desired, a CPU. The Merlin Compiler assumes a distributed memory model, which means that CPUs and FPGAs have their own memory space. The data between the different memory spaces are transferred via a PCIe connection. The focus of the Merlin Compiler is on automating the entire process at the cost of more fine grained control possible by the engineer.

Spearmint [14] is a source-to-source translator that translates annotated C/C++ code to parallelized CUDA C/C++ code for a CPU-GPU system using the LLVM compiler infrastructure. The annotated C/C++ code can consist of five different types of pragmas. A modified LLVM Clang tooling library is created to handle the newly defined pragmas, as pragmas are automatically removed from the AST. The Spearmint framework uses this tool to traverse the AST and replace the annotated code using LLVMs FrontendAction.

The Spearmint project is a continuation on the Mint [15] project. which used the Rose Compiler infrastructure, that has support for a mutable AST. The Mint project changes the AST to change the architecture of the software. The issue with directly using the AST provided by the Rose Compiler infrastructure is that it is very complex, thus requiring a huge amount of coding effort to maintain. The move to LLVM in the Spearmint project reduces this, because LLVM has facilities (FrontendAction, RecursiveASTVisitor) that allow for source-to-source translation.

Examples that apply the DAE architecture on code written for HLS tools have been explored in the past. In the bioinformatics field the DAE architecture is used to create an FPGA accelerator for detection of positive selection in large-scale single-

nucleotide polymorphisms data [16] [17]. It showed an increased speedup when compared to software tools varying from 20x to 751x. The reason for the large speedup of this accelerator compared to the software tools was due to using an algorithm that exploits the high degree of parallelism of FPGAs.

3.2 DAE frameworks

This section describes what relevant works have been researched in previous works that specifically use the DAE architecture.

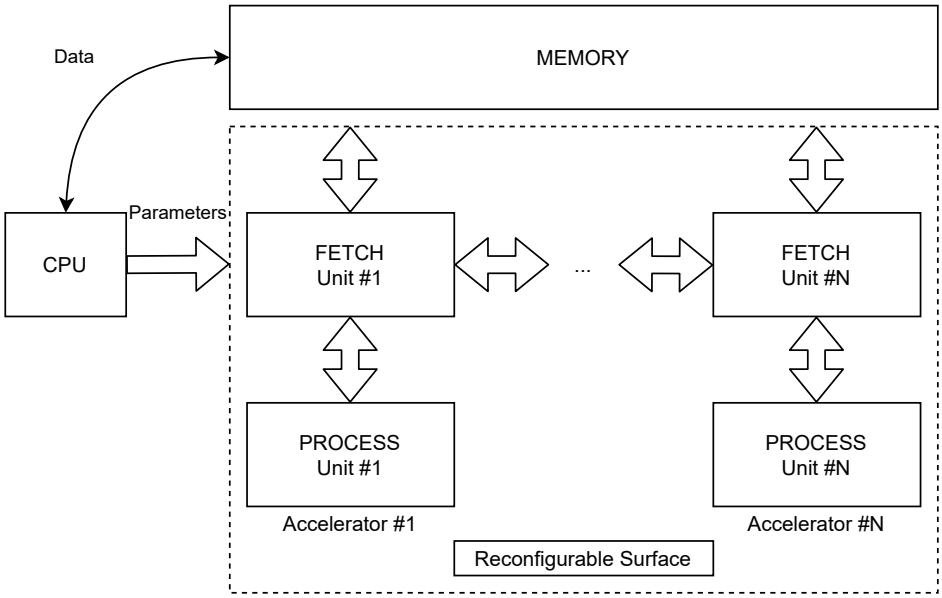


Figure 3.2: Decoupled Access-Execute architecture for Reconfigurable accelerators adapted from [2]

The Decoupled Access-Execute architecture and framework for Reconfigurable accelerators [2] increases the speedup of an application by expanding the capabilities of the DAE architecture. It specifically targets hybrid systems with one or more CPUs and FPGAs.

Figure 3.2 gives an overview of the architecture. It consists of multiple fetch units and processing units, the naming is analogous to the access and execute units described by the DAE architecture. There is a fetch unit that is connected to the CPU and memory, where the connection to the CPU is needed for passing program parameters (start memory addresses, etc.) and the input and output data from the program is handled by the memory connection. The fetch unit can also access data from other accelerators in the reconfigurable system. The processing unit performs all logic and arithmetic operations. The different units are interconnected using first in first out (FIFO) queues.

The resulting framework is evaluated using three different benchmarks: general matrix multiplication (gemm), sparse matrix/vector multiplication (spmv) and the Needleman-Wunsch algorithm. The results are compared against an unoptimized HLS implementation, which only has basic data I/O optimizations. Then its results are evaluated against an optimized HLS implementation that, in addition to the unoptimized, has design specific optimization directives (pipelining, unrolling). The DAER HLS implementation is constructed from the optimized HLS implementation with the key change of changing a target section with a DAE transformed version. On average, the proposed architecture achieves a speedup of 2x compared against the baseline HLS versions due to more efficient data accessing.

The main downside of this framework is that the proposed framework requires manually changing the entire structure of the source code.

The work described by Chen and Suh [6] uses the DAE architecture to improve the speedup of algorithms at the cost of area. They observed that the access part must run faster in the DAE architecture compared to the non-DAE architecture otherwise there would be no improvement in the speedup of the algorithm. In addition to having access and execute units a memory unit is added that behaves as a proxy though which memory accessing is handled. The memory unit is responsible for memory request handling and data forwarding while the access unit remains responsible for address generation and sending memory requests.

In addition to applying the DAE architecture, a prefetcher is implemented to further increase the potential speedup achievable. When only applying the DAE architecture the observed speedup is 1.89x while adding prefetching increased the speedup up to 2.28x.

The main downside here is that this work focuses only on optimizing the speedup. This thesis has an additional focus on readability such that the engineer can experiment or perform further optimizations to the transformed algorithm.

CASCADE [18] is a novel Decoupled Access-Execute CGRA design. A CGRA is, by design, an array of Processing Elements (PEs). Most of these PEs are allocated to Address Generation Instructions (AGIs) in a kernel. The percentage of AGIs used can range from 20% to 80% depending on when the CGRA uses single-bank or multi-bank memory.

CASCADE proposes to decouple the address generation to custom designed programmable hardware. This makes the CGRA focus purely on the computation, while address generation is handled by specialized hardware (Stream Engine). An ideal decoupled access-execute CGRA has an on average 5x increase in throughput compared to an ideal conventional CGRA. The LLVM framework is used to provide a complete end-to-end solution to compile code to a configuration for the CGRA and the Stream Engine.

Chapter 4

Framework design

This chapter describes the design of the framework. The system that this thesis targets is described in Section 4.1. The complete overview of the individual steps of the framework is described in Section 4.2. Section 4.3 describes how memory accessing elements are located. The dependencies of the memory accessing elements are solved in Section 4.4. Section 4.5 describes how communication and synchronization is facilitated between the different units that run in parallel. The code transformations are not directly applied on the input unparsed code. Instead, they are applied on an IR the design of which is described in Section 4.6. Section 4.7 shows a demonstration how the framework will apply the methods described on how to translate C/C++ source code to a DAE version. Section 4.8 describes the limitations that were identified as a result of varying methods to access data from memory and how it is handled by the rest of the source code.

4.1 System architecture

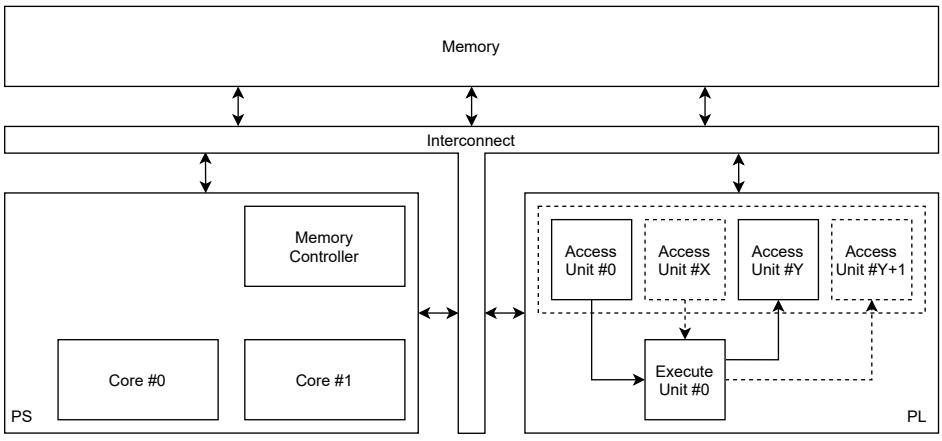


Figure 4.1: Targeted system architecture

This thesis targets a system which consists of two separate components: A Processor System (PS) and a Programmable Logic (PL). They can be either all on a single chip (System on a chip (SoC)) or completely separate using an external interconnect. The architecture of this system is depicted in Figure 4.1. The PS contains multiple processors and an external memory interface controller. The PL contains the hardware accelerators, in this case an example accelerator is shown that conforms the DAE architecture.

Due to area, power and performance constraints the algorithm may be partially synthesized to PL while the other part can run on the PS, the PS will wait for the PL until it has completed the relevant part of the algorithm.

4.2 Framework overview

Figure 4.2 shows the general overview of the framework. It essentially consists of five distinct steps. The framework first needs to parse the C/C++ code to the IR as it was described in the previous chapter. The following phase is on the extraction of the different DAE units, optimizing that for high-level synthesis and writing a C/C++ source file that the HLS tool will use to synthesize to a hardware description.

During parsing the developer selects a target block of code to be transformed to the DAE architecture and prepared for the HLS tool. The selected target code is parsed into the IR. The AST is used in conjunction with tokens to obtain all the information needed to parse the code into the IR. The AST is using the preprocessed C/C++ source code, while tokens represent the text as it is written in the source files instead of being preprocessed.

After parsing the memory accessing elements are identified using the algorithms defined in the previous chapter. After identification the memory accessing elements are used to create the access units. The previously parsed IR represents the execute unit. Another step is needed to replace the memory accessing elements from the execute unit, it is replaced by stream links that connect to the access units.

Optimization consists of automatically inserting directives for use with the HLS tool. Loop pipelining, setting the correct interfaces, local array partitioning and enabling parallel tasks are part of this phase. This results in code that is efficient for HLS tools. The intermediate representation needs to be converted back to C/C++ source files so that that HLS tool can use it. It consists of writing tokens to a source file making sure that spaces are inserted whenever necessary. The HLS tool is used to finally synthesize the source files to an RTL description.

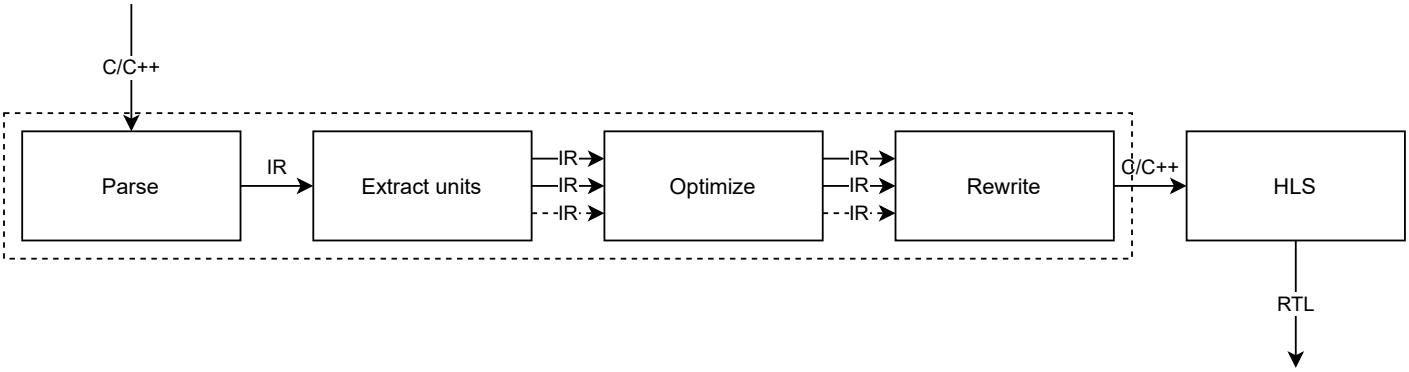


Figure 4.2: The flow of the framework

4.3 Memory accessing patterns

The key factor in the DAE architecture, described in Section 2.3, is the extraction of memory elements. Suppose that the top-level function *vector_adder* described in Listing 4.1 is targeted for high-level synthesis. This function contains multiple loops to calculate $M * \vec{1}$. In other words a matrix M is multiplied with a all-ones vector.

In this example there are two memory accessing elements, one input matrix *m1* and one output vector *v1*.

These memory accessing elements can be located by analyzing how they are structured. In this case the variable has a name followed by two brackets and a number in between. Another structure that represents a memory accessing pattern is dereferencing of pointers (**v1*). Algorithm 1 shows how memory accessing patterns are located, in this thesis only the first memory accessing pattern is supported. The memory accesses are stored in a separate list to be used at a later stage.

```
1  void vector_adder(int *m1, int *v1) {
2      int i, j;
3      int i_row;
4      int sum;
5
6      for(i=0; i<col_size; i++) {
7          i_row = i * row_size;
8          sum = 0;
9          for(j=0; j<row_size; j++) {
10             sum += m1[i_row + j];
11         }
12         v1[i] = sum;
13     }
14 }
```

Listing 4.1: Matrix vector addition

Algorithm 1 Memory access pattern locating

Precondition: $source$ as the algorithmic description

```
1: function MEMORY_ACCESS_LOCATOR(source)
2:     mem_elem ← ∅                ▷ mem_elem: List of memory accessing statements
3:     for all statement ∈ source do
4:         if square_bracket in statement then
5:             if integer in between square_bracket then
6:                 mem_elem ← mem_elem||statement
7:             end if
8:         end if
9:     end for
10:    return mem_elem
11: end function
```

4.4 Unit creation

Next to the identification of the accessing elements there are the address generation of the memory accessing element and the frequency at which it is accessed. As mentioned in Section 2.3 this thesis uses multiple access units and a single execute unit. The execute unit is behaviourally identical to the input code with the exception of the memory accessing element replaced by a stream that is connected to an access unit.

Consider still the same code snippet depicted in Listing 4.1. The address generation is in that case handled by the pointer index of the memory accessing elements: *i_row + j*, needed for the *m1* memory access and *i*, needed for the *v1* memory access.

At this point a reverse copy of the IR is made at the location where a memory accessing statement is found. Algorithm 2 describes how this reverse copy of statements is created, all statements listed after the memory accessing statement are ignored as the head (the current location in the list of statements) is located at the memory accessing statement. It traverses the list of statements in reverse prepending statements onto the new unit.

The access unit created now has the complete structure of the final unit, the memory accessing elements are accessed the same number of times as the input C/C++ code. The statements that aren't relevant for address generation are removed as the access unit copy algorithm didn't take into account the dependencies. Algorithm 3 describes how this is achieved. It starts from the head, the memory

Algorithm 2 Access unit creation

Precondition: $source\_statement$ The complete tree statement structure pointing to the memory access

```
1: function REVERSE_COPY(source_statement)
2:     unit ← Copy of the head at source_statement
3:     access_head ← Copy of the head at unit
4:     for previous statements in source_statement do
5:         unit ← Prepend a copy of the current statement
6:         unit ← Previous unit statement
7:     end for
8:     return access_head
9: end function
```

accessing statement, and traverses the list of statements backwards to followed by a check if the statement is used by future statements. If not, it will be removed from the statement set.

The created access unit needs the statement that contains the memory accessing element altered such that it reads or writes the data from memory into a stream that is connected to the execute unit. Algorithm 4 is used to identify if the memory accessing element is reading from memory or writing to memory. The entire statement containing the memory accessing element is replaced by either a read or a write from memory connected to a stream. A stream has a single input and a single output meaning that it is connected to a single access unit and a single execute unit.

4.5 Unit communication and synchronization

An important aspect of the DAE architecture is that all units run in parallel. The DAE architecture uses stream queues to link the different units. Any unit that requests data yet to become available is stalled until data becomes available. This allows the different unit to run in parallel and at different speeds while never losing synchronization. The responsibility for this stalling and synchronization is handled by the queues as it's the connecting element.

As discussed in Section 4.1 an external interface between the units and memory is used. In this thesis the external AXI4 interface protocol is used. This also has the ability to perform burst reads and writes, potentially allowing for a higher throughput. HLS controls whether a burst read or write will be enabled for an interface as it depends on the code structure. For instance, a simple loop that reads from a memory

Algorithm 3 Dependence elimination

Precondition: $access\_unit$ A partial copy of the algorithmic description with the access at its head.

```
1: function DEPENDENCE_ELIMINATION(access_unit)
2:     access_head ← The head of the access_unit
3:     while access_head has previous statement do
4:         if access_head has no uses in next statements then
5:             tmp ← access_head
6:         end if
7:         access_head ← Previous statement in access_unit
8:         if tmp is set then
9:             tmp ← Remove statement
10:        end if
11:    end while
12: end function
```

Algorithm 4 Memory access read or write identification

Precondition: $statement$ Statement at which the memory accessing element is located.

```
1: function ACCESS_READ_OR_WRITE(statement)
2:     if Has equals after memory accessing element then
3:         return write access
4:     else
5:         return read access
6:     end if
7: end function
```

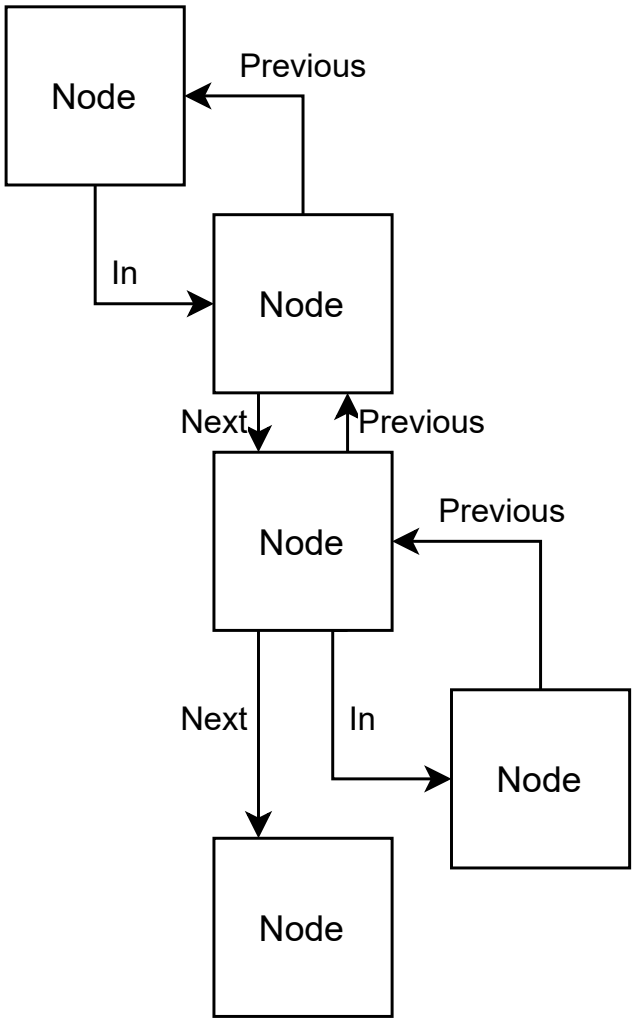
location that increases with a fixed size will have burst reads or writes, but more complex memory accessing patterns might not.

4.6 The intermediate representation

The DAE architecture does not change the actual amount of data fetching, otherwise issues might occur where not enough data is fetched resulting in units waiting for data that will never become available. In source-to-source translation, see Section 2.4, abstraction in the form of an intermediate representation is built from the source code. Transformations are applied in further phases in a compiler infrastruc-

ture.

For compiler infrastructures a low-level abstract internal intermediate representation of the source code is used. This abstract form is not desirable as information such as variable names and/or loop structures may be lost. This means that an internal representation is to be used that still contains the high-level information from the source code while also providing the ability to transform that into the DAE architecture.



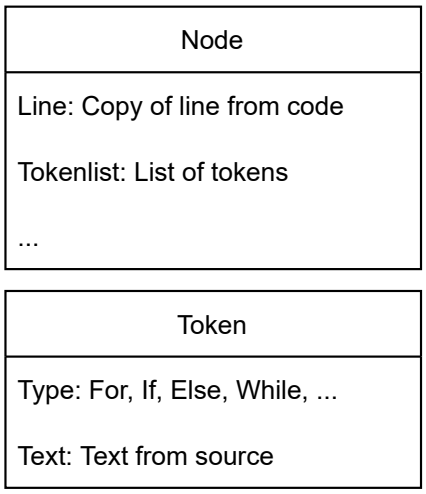


Figure 4.4: Contents of a node and tokens

Figure 4.3: The intermediate representation

Figure 4.3 shows the overview of the designed IR. Any given node has a relation to other nodes. This is described by the *previous*, *next* and *in* edges. Each node has a node on the previous edge with the exception of the root node. Each node can have up to two succeeding nodes: *next* and/or *in*. It is also possible for it to have no consequent nodes. A *next* edge describes that the following node has a C/C++ statement that is evaluated after the previous node. An *in* edge implies that the following node is the body of the previous node. This could be for example the

body of an *if* statement or a *for* loop.

The contents of each *node*, shown in Figure 4.4, is structured such that all information from the original code is encompassed within it. One of the goals is to have the output C/C++ code be human readable, as such there is dedicated *line* field that holds the line statement as it was written in the input C/C++ code. This ensures that the input code is regenerated when no code transformation to be performed. It is implied that this field is read-only, meaning that any code transformation should not be applied to this line but instead a different field. The *tokenlist* is used for manipulation. This field contains essentially the same contents as the *line* field, but structured as a linked list that can be safely manipulated. A *token* has the text from the source code associated with it and a type.

The IR should be rewritten to a C/C++ source file. A dedicated module is used for this. Based on the *line* or *tokenlist* it can build the source file. The *tokenlist* is always preferred over the *line*. The formatting of the tokens into code is also the responsibility of this module.

4.7 Transformation example

```
1  stencil_label1:for (r=0; r<row_size-2; r++) {
2      stencil_label2:for (c=0; c<col_size-2; c++) {
3          temp = (TYPE)0;
4          stencil_label3:for (k1=0;k1<3;k1++){
5              stencil_label4:for (k2=0;k2<3;k2++){
6                  mul = filter[k1*3 + k2] * orig[(r+k1)*col_size + c+k2];
7                  temp += mul;
8              }
9          }
10         sol[(r*col_size) + c] = temp;
11     }
12 }
```

Listing 4.2: Example input code

To demonstrate how a piece of code is transformed into the DAE architecture the stencil2d benchmark, part of the MachSuite benchmark suite, is used. Listing 4.2 shows the most relevant code snippet of the algorithm, it consists of four loops and three external memory accesses via pointers. The complete code of this example is available in Appendix B.3. Using the AST the structure of the code is extracted. The most relevant parts of the AST as used by LLVM is shown in Figure 4.5. This AST is transformed into the intermediate representation. Even though the AST and

IR present a tree-like structure, they are not interchangeable. The AST evaluates every individual statement, this is not needed for the IR as it contains structural information of loops and conditional statements, everything else is left as is and stored as a complete statement within a node. For instance, a for-loop would have nested statements in the IR (an *in* edge), while an assignment shouldn't have nested statements. This is in contrast to the AST where a statement can be composed of multiple statements. In LLVM an expression is a subset of a statement, making a statement consist of other statements. Figure 4.6 shows the intermediate representation that is generated from the AST. Each node has a node name for explanation purposes.

Now that the code is in the IR form the next step is to extract the memory accessing elements according to Algorithm 1. Starting from the root of the IR the nodes are recursively traversed. On each node the tokens are compared. Once the first *[* is found it can start extracting the memory accessing element. The first *[* is found in *node6*. The token defined before the *[* token is the variable name. To verify correct code the rest of the tokens are also compared until a *]* is found. A reference to this node is appended to a list that holds all memory accessing elements. Traversing the entire IR results in the list of memory accessing elements holding nodes with references to the following memory accesses: *filter*, *orig* and *sol*.

Next up is the creation of access units based on the memory accessing elements. Considering that the previously defined list contains references, the IR can still be used to traverse from a given point. Using Algorithm 2 a new subset of the IR is created by reverse traversing the IR at the current head (i.e. the memory accessing element). This will form the base of an access unit. Using context information, the placement of where the memory accessing element, it is determined if the memory accessing element is a read or write memory accessing element. Listing 4.2 shows that *filter* and *orig* are read from while *sol* is written to, this information is purely based on the positioning of the memory element in relation to the equals sign. This information is also used in the execute unit to replace the memory accessing elements with the stream that connects to the access units. The resulting IR of the access unit for the *filter* memory accessing element is shown in Figure 4.7.

Figure 4.8 shows the final architecture on how the units are connected.

The goal of this thesis is to increase the speedup while also having the resulting code from the framework in a readable form. To increase the speedup automatically all loops are pipelined with an iteration interval of one. Meaning that the HLS tool should try as best as it can to reduce the iteration interval of the outside loop to one. This might not be possible if memory is accessed multiple times in the same loop cycle.

```
LabelStmt 0x5576d3b3e4b8 <line:15:5, line:26:5> 'stencil_label1'
`-ForStmt 0x5576d3b3e430 <line:15:20, line:26:5>
  |-BinaryOperator 0x5576d3b3c6a0 <line:15:25, col:27> 'int' lvalue '='
  | |-DeclRefExpr 0x5576d3b3c660 <col:25> 'int' lvalue Var 0x5576d3b3c318 'r' 'int'
  | `-IntegerLiteral 0x5576d3b3c680 <col:27> 'int' 0
  |-<<<NULL>>>
  |-BinaryOperator 0x5576d3b3c758 <col:30, col:41> 'bool' '<'
  | |-ImplicitCastExpr 0x5576d3b3c740 <col:30> 'int' <LValueToRValue>
  | | `-DeclRefExpr 0x5576d3b3c6c0 <col:30> 'int' lvalue Var 0x5576d3b3c318 'r' 'int'
  | `-BinaryOperator 0x5576d3b3c720 <./stencil.h:7:18, stencil.cpp:15:41> 'int' '-'
  |   |-IntegerLiteral 0x5576d3b3c6e0 <./stencil.h:7:18> 'int' 128
  |   `-IntegerLiteral 0x5576d3b3c700 <stencil.cpp:15:41> 'int' 2
  |-UnaryOperator 0x5576d3b3c798 <col:44, col:45> 'int' postfix '++'
  | `-DeclRefExpr 0x5576d3b3c778 <col:44> 'int' lvalue Var 0x5576d3b3c318 'r' 'int'
  `-CompoundStmt 0x5576d3b3e418 <col:49, line:26:5>
    `-LabelStmt 0x5576d3b3e400 <line:16:9, line:25:9> 'stencil_label2'
      `-ForStmt 0x5576d3b3e378 <line:16:24, line:25:9>
        |- ...
        `-CompoundStmt 0x5576d3b3e350 <col:53, line:25:9>
          |-BinaryOperator 0x5576d3b3c978 <line:17:13, col:26> 'int32_t':'int' lvalue '='
          | |-DeclRefExpr 0x5576d3b3c900 <col:13> 'int32_t':'int' lvalue Var 0x5576d3b3c550 'temp' 'int32_t':'int'
          | `-CStyleCastExpr 0x5576d3b3c950 <col:20, col:26> 'int32_t':'int' <NoOp>
          |   `-IntegerLiteral 0x5576d3b3c920 <col:26> 'int' 0
          |-LabelStmt 0x5576d3b3e198 <line:18:13, line:23:13> 'stencil_label3'
          | `-ForStmt 0x5576d3b3e110 <line:18:28, line:23:13>
          |   |- ...
          |   `-CompoundStmt 0x5576d3b3e0f8 <col:48, line:23:13>
          |     `-LabelStmt 0x5576d3b3e0e0 <line:19:17, line:22:17> 'stencil_label4'
          |       `-ForStmt 0x5576d3b3d010 <line:19:32, line:22:17>
          |         |-BinaryOperator 0x5576d3b3cae8 <line:19:37, col:40> 'int' lvalue '='
          |         | |-DeclRefExpr 0x5576d3b3caa8 <col:37> 'int' lvalue Var 0x5576d3b3c498 'k2' 'int'
          |         | `-IntegerLiteral 0x5576d3b3cac8 <col:40> 'int' 0
          |         |-<<<NULL>>>
          |         |-BinaryOperator 0x5576d3b3cb60 <col:42, col:45> 'bool' '<'
          |         | |-ImplicitCastExpr 0x5576d3b3cb48 <col:42> 'int' <LValueToRValue>
          |         | | `-DeclRefExpr 0x5576d3b3cb08 <col:42> 'int' lvalue Var 0x5576d3b3c498 'k2' 'int'
          |         | `-IntegerLiteral 0x5576d3b3cb28 <col:45> 'int' 3
          |         |-UnaryOperator 0x5576d3b3cba0 <col:47, col:49> 'int' postfix '++'
          |         | `-DeclRefExpr 0x5576d3b3cb80 <col:47> 'int' lvalue Var 0x5576d3b3c498 'k2' 'int'
          |         `-CompoundStmt 0x5576d3b3cff0 <col:52, line:22:17>
          |           |-BinaryOperator 0x5576d3b3cf48 <line:20:21, col:74> 'int32_t':'int' lvalue '='
          |           | |-DeclRefExpr 0x5576d3b3cbb8 <col:21> 'int32_t':'int' lvalue Var 0x5576d3b3c5c8 'mul' 'int32_t':'int'
          |           | `-BinaryOperator 0x5576d3b3cf28 <col:27, col:74> 'int' '*'
          |           |   |-ImplicitCastExpr 0x5576d3b3cef8 <col:27, col:43> 'int32_t':'int' <LValueToRValue>
          |           |   | `-ArraySubscriptExpr 0x5576d3b3cce0 <col:27, col:43> 'int32_t':'int' lvalue
          |           |   |   |-ImplicitCastExpr 0x5576d3b3ccc8 <col:27> 'int32_t *' <LValueToRValue>
          |           |   |   | `-DeclRefExpr 0x5576d3b3cbd8 <col:27> 'int32_t *' lvalue ParmVar 0x5576d3b3c160 'filter' 'int32_t *'
          |           |   |   `-BinaryOperator 0x5576d3b3cca8 <col:34, col:41> 'int' '+'
          |           |   |     |-BinaryOperator 0x5576d3b3cc50 <col:34, col:37> 'int' '*'
          |           |   |     | |-ImplicitCastExpr 0x5576d3b3cc38 <col:34> 'int' <LValueToRValue>
          |           |   |     | | `-DeclRefExpr 0x5576d3b3cbf8 <col:34> 'int' lvalue Var 0x5576d3b3c418 'k1' 'int'
          |           |   |     | `-IntegerLiteral 0x5576d3b3cc18 <col:37> 'int' 3
          |           |   |     `-ImplicitCastExpr 0x5576d3b3cc90 <col:41> 'int' <LValueToRValue>
          |           |   |       `-DeclRefExpr 0x5576d3b3cc70 <col:41> 'int' lvalue Var 0x5576d3b3c498 'k2' 'int'
          |           |   `-ImplicitCastExpr 0x5576d3b3cf10 <col:47, col:74> 'int32_t':'int' <LValueToRValue>
          |           |     `-ArraySubscriptExpr 0x5576d3b3ced8 <col:47, col:74> 'int32_t':'int' lvalue
          |           |       |-ImplicitCastExpr 0x5576d3b3cec0 <col:47> 'int32_t *' <LValueToRValue>
          |           |       | `-DeclRefExpr 0x5576d3b3cd00 <col:47> 'int32_t *' lvalue ParmVar 0x5576d3b3c070 'orig' 'int32_t *'
          |           |       `-BinaryOperator 0x5576d3b3cea0 <col:52, col:72> 'int' '+'
          |           |         |-BinaryOperator 0x5576d3b3ce48 <col:52, col:70> 'int' '+'
          |           |         | |-BinaryOperator 0x5576d3b3cdf0 <col:52, ./stencil.h:6:18> 'int' '*'
          |           |         | | |-ParenExpr 0x5576d3b3cdb0 <stencil.cpp:20:52, col:57> 'int'
          |           |         | | | `-BinaryOperator 0x5576d3b3cd90 <col:53, col:55> 'int' '+'
          |           |         | | |   |-ImplicitCastExpr 0x5576d3b3cd60 <col:53> 'int' <LValueToRValue>
          |           |         | | |   | `-DeclRefExpr 0x5576d3b3cd20 <col:53> 'int' lvalue Var 0x5576d3b3c318 'r' 'int'
          |           |         | | |   `-ImplicitCastExpr 0x5576d3b3cd78 <col:55> 'int' <LValueToRValue>
          |           |         | | |     `-DeclRefExpr 0x5576d3b3cd40 <col:55> 'int' lvalue Var 0x5576d3b3c418 'k1' 'int'
          |           |         | | `-IntegerLiteral 0x5576d3b3cdd0 <./stencil.h:6:18> 'int' 64
          |           |         | `-ImplicitCastExpr 0x5576d3b3ce30 <stencil.cpp:20:70> 'int' <LValueToRValue>
          |           |         |   `-DeclRefExpr 0x5576d3b3ce10 <col:70> 'int' lvalue Var 0x5576d3b3c398 'c' 'int'
          |           |         `-ImplicitCastExpr 0x5576d3b3ce88 <col:72> 'int' <LValueToRValue>
          |           |           `-DeclRefExpr 0x5576d3b3ce68 <col:72> 'int' lvalue Var 0x5576d3b3c498 'k2' 'int'
          |           `-CompoundAssignOperator 0x5576d3b3cfc0 <line:21:21, col:29> 'int32_t':'int' lvalue '+=' ComputeLHSTy='int' ComputeResultTy='int'
          |             |-DeclRefExpr 0x5576d3b3cf68 <col:21> 'int32_t':'int' lvalue Var 0x5576d3b3c550 'temp' 'int32_t':'int'
          |             `-ImplicitCastExpr 0x5576d3b3cfa8 <col:29> 'int32_t':'int' <LValueToRValue>
          |               `-DeclRefExpr 0x5576d3b3cf88 <col:29> 'int32_t':'int' lvalue Var 0x5576d3b3c5c8 'mul' 'int32_t':'int'
          `-BinaryOperator 0x5576d3b3e330 <line:24:13, col:37> 'int32_t':'int' lvalue '='
            |-ArraySubscriptExpr 0x5576d3b3e2d8 <col:13, col:33> 'int32_t':'int' lvalue
            | |-ImplicitCastExpr 0x5576d3b3e2c0 <col:13> 'int32_t *' <LValueToRValue>
            | | `-DeclRefExpr 0x5576d3b3e1b0 <col:13> 'int32_t *' lvalue ParmVar 0x5576d3b3c0e8 'sol' 'int32_t *'
            | `-BinaryOperator 0x5576d3b3e2a0 <col:17, col:32> 'int' '+'
            |   |-ParenExpr 0x5576d3b3e248 <col:17, col:28> 'int'
            |   | `-BinaryOperator 0x5576d3b3e228 <col:18, ./stencil.h:6:18> 'int' '*'
            |   |   |-ImplicitCastExpr 0x5576d3b3e210 <stencil.cpp:24:18> 'int' <LValueToRValue>
            |   |   | `-DeclRefExpr 0x5576d3b3e1d0 <col:18> 'int' lvalue Var 0x5576d3b3c318 'r' 'int'
            |   |   `-IntegerLiteral 0x5576d3b3e1f0 <./stencil.h:6:18> 'int' 64
            |   `-ImplicitCastExpr 0x5576d3b3e288 <stencil.cpp:24:32> 'int' <LValueToRValue>
            |     `-DeclRefExpr 0x5576d3b3e268 <col:32> 'int' lvalue Var 0x5576d3b3c398 'c' 'int'
            `-ImplicitCastExpr 0x5576d3b3e318 <col:37> 'int32_t':'int' <LValueToRValue>
              `-DeclRefExpr 0x5576d3b3e2f8 <col:37> 'int32_t':'int' lvalue Var 0x5576d3b3c550 'temp' 'int32_t':'int'
```

Figure 4.5: AST of the example code

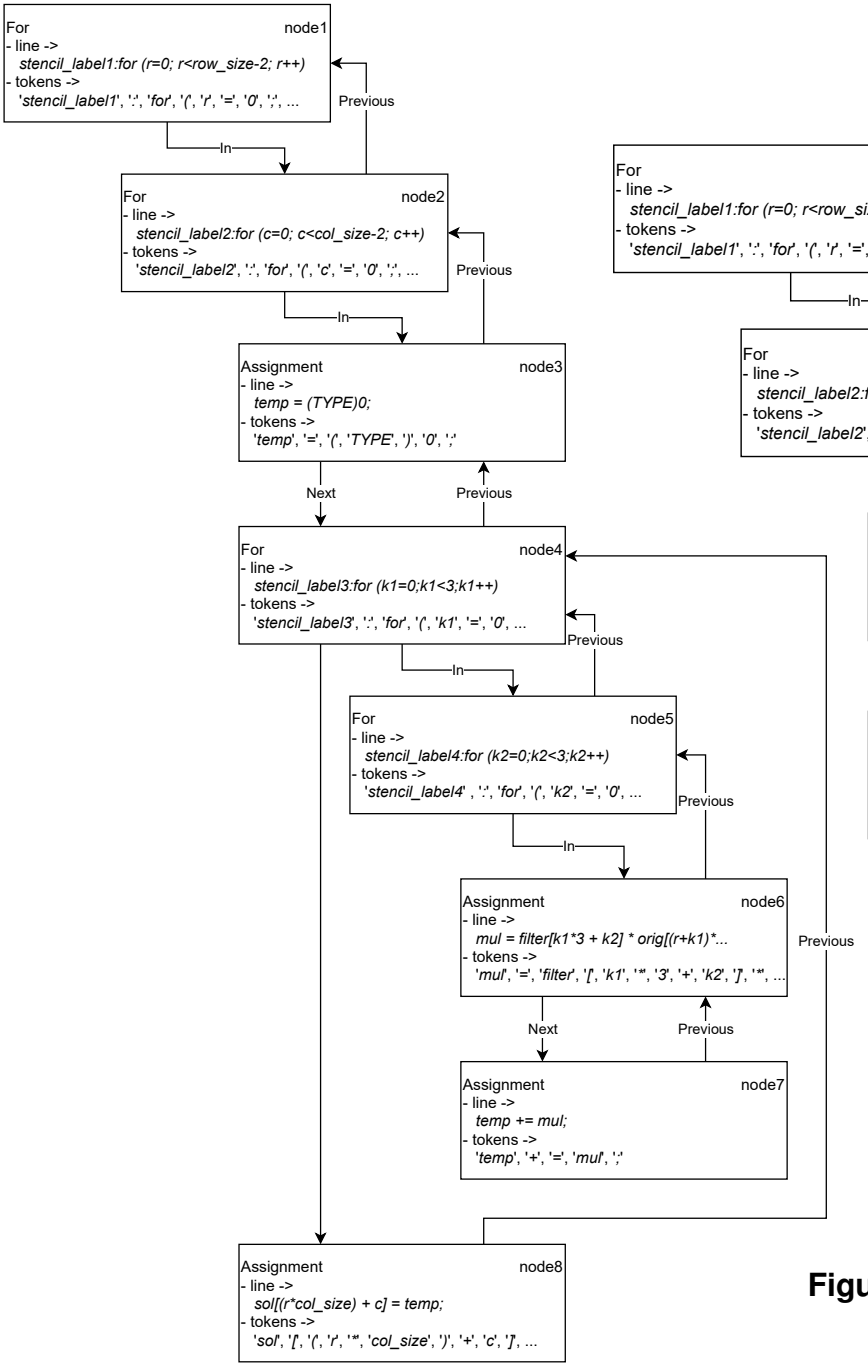


Figure 4.7: IR of the access unit

Figure 4.6: IR of the example code

Now that all intermediate representations are created that represent the different units the final step is to convert it to source code that the HLS tool can use. The code for connecting the units is shown in Listing 4.3. A type *stream_t* is used to define streams that connect the units. An important aspect here is the usage of the *HLS DATAFLOW* pragma, as this allows the units to run in parallel.

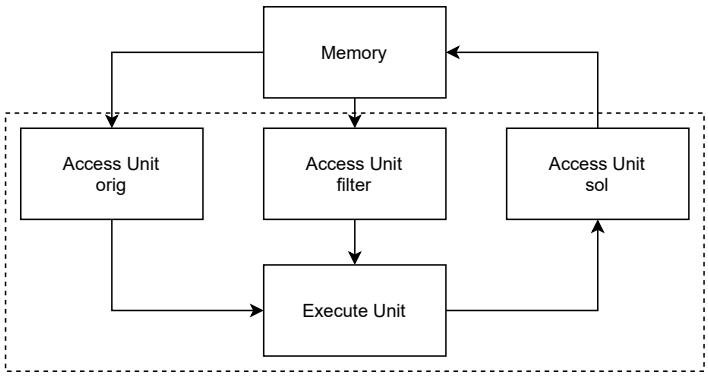


Figure 4.8: Complete architecture of example

```
1  stream_t filter_rs0;
2  #pragma HLS STREAM variable=filter_rs0
3  stream_t orig_rs1;
4  #pragma HLS STREAM variable=orig_rs1
5  stream_t sol_ws2;
6  #pragma HLS STREAM variable=sol_ws2
7  #pragma HLS DATAFLOW
8  fetch_unit0(filter, filter_rs0);
9  fetch_unit1(orig, orig_rs1);
10 process_unit(filter_rs0, orig_rs1, sol_ws2);
11 write_unit2(sol, sol_ws2);
```

Listing 4.3: Final code for connecting the units

4.8 Current limitations

The transformation to the DAE architecture depends on the structure of the input C/C++ code. The algorithms defined in the previous chapter do not encapsulate all possible memory accessing patterns. This section shows the patterns that are not supported and how to manually adapt the code such that it conforms to the DAE architecture.

Consider the code in Listing 4.4. Whenever a memory accessing element is accessed multiple times an issue is introduced as every usage of a memory accessing element results in a new access unit.

The generated DAE structure of the code is shown in Listing 4.5. There are two separate access units created for the *m1* memory accessing element, one for *m1[0]* and one for *m1[j]*. While that code is not problematic the issue is with connecting the units and letting them run in parallel. The *m_access* function connects the units and using the *HLS DATAFLOW* pragma all of them are run in parallel. This does introduce the issue where *m1* is accessed in parallel by *fetch_unit0* and *fetch_unit1*

this is not supported as *m1* is synthesized as a single port.

A solution to this issue is to either create two ports for the same memory accessing element or to merge the units such that a memory accessing port is connected to one access unit. The former has the advantage that it can achieve a higher speed as the two ports can operate independently. Listing 4.6 shows the solution of using two differently named memory accessing elements. When calling the accelerator, different parameters should be prepared such that *m1* is supplied to both *m01* and *m11*.

```
1  void m_access(int *m1, int *out) {
2      int i, j;
3      int mult;
4
5      outer:for(i=0;i<10;i++) {
6          mult = m1[0];
7          middle:for(j=1;j<10;j++) {
8              mult = mult + m1[j];
9          }
10         out[i] = mult;
11     }
12 }
```

Listing 4.4: Multiple accesses of the same pointer

```
1  void fetch_unit0(int * m1, hls::stream<int> & m1_rs0) {
2    int i;
3    fetch_unit0_outer: for(i = 0; i<10; i ++) {
4  #pragma HLS PIPELINE
5      m1_rs0.write(m1[0]);
6    }
7  }
8  void fetch_unit1(int * m1, hls::stream<int> & m1_rs1) {
9    int i;
10   int j;
11   fetch_unit1_outer: for(i = 0; i<10; i ++) {
12 #pragma HLS PIPELINE
13     fetch_unit1_middle: for(j = 1; j<10; j ++) {
14 #pragma HLS PIPELINE
15       m1_rs1.write(m1[j]);
16     }
17   }
18 }
19 void write_unit2(int * out, hls::stream<int> & out_ws2) {
20   int i;
21   write_unit2_outer: for(i = 0; i<10; i ++) {
22 #pragma HLS PIPELINE
```

```
23      out[i] = out_ws2.read();
24    }
25  }
26  void process_unit(hls::stream<int> & m1_rs0, hls::stream<int> & m1_rs1,
    ↪  hls::stream<int> & out_ws2) {
27    int i;
28    int mult;
29    int j;
30    outer: for(i = 0; i<10; i ++) {
31  #pragma HLS PIPELINE
32      mult = m1_rs0.read();
33      middle: for(j = 1; j<10; j ++) {
34  #pragma HLS PIPELINE
35        mult = mult + m1_rs1.read();
36      }
37      out_ws2.write( mult);
38    }
39  }
40  void m_access(int * m1, int * out) {
41  #pragma HLS INTERFACE s_axilite port=m1
42  #pragma HLS INTERFACE s_axilite port=out
43  #pragma HLS INTERFACE s_axilite port=return
44  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0 port=m1
45  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem1 port=out
46    hls::stream<int> m1_rs0;
47  #pragma HLS STREAM variable=m1_rs0
48    hls::stream<int> m1_rs1;
49  #pragma HLS STREAM variable=m1_rs1
50    hls::stream<int> out_ws2;
51  #pragma HLS STREAM variable=out_ws2
52  #pragma HLS DATAFLOW
53    fetch_unit0(m1, m1_rs0);
54    fetch_unit1(m1, m1_rs1);
55    process_unit(m1_rs0, m1_rs1, out_ws2);
56    write_unit2(out, out_ws2);
57  }
```

Listing 4.5: DAE structure of multiple accesses of the same pointer

The units work independently of one another, this introduces another issue when a memory accessing element is used multiple times but one stores data while the other reads. Consider the code in Listing 4.7 it uses the same memory accessing element for storing the data. Assume that *m1* is an array in memory of size ten. Essentially every iteration it calculates the sum of all elements then stores it again in the array. This results in the elements in the array growing. So if the sequence is initialized to: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ the resulting sequence will become

```
1  void m_access(int *m01, int *m11, int *out) {
2      int i, j, k;
3      int mult;
4
5      outer:for(i=0;i<10;i++) {
6          mult = m01[0];
7          middle:for(j=1;j<10;j++) {
8              mult = mult + m11[j];
9          }
10         out[i] = mult;
11     }
12 }
```

Listing 4.6: Solution to multiple accesses of the same pointer

$55, 109, 216, 429, 854, 1703, 3400, 6793, 13578, 27147$. However when transforming this the the DAE architecture units lose synchronization as the access unit for reading from memory does not depend on the access unit that writes to memory. As a result 'old' values from memory can be used in next iterations.

To solution this issue, the same solution used for the previous issue can be used (i.e. move all logic to a single accessing unit). Another method that is proposed by Smith [10] is to have a stream for memory addresses and check if a memory store for this address is already in that stream and stall fetching if it does exist. Listing 4.8 shows the former solution in the transformed code.

```
1  void read_write(int *m1) {
2      int i, j, k;
3      int mult;
4
5      outer:for(i=0;i<10;i++) {
6          mult = 0;
7          middle:for(j=0;j<10;j++) {
8              mult = mult + m1[j];
9          }
10         m1[i] = mult;
11     }
12 }
```

Listing 4.7: Read after write limitation

```
void fetch_unit0(int * m1, hls::stream<int> & m1_rs0, hls::stream<int> & m1_ws1) {
  int i;
  int mult;
  int j;
  fetch_unit0_outer: for(i = 0; i<10; i ++) {
#pragma HLS PIPELINE
    fetch_unit0_middle: for(j = 0; j<10; j ++) {
#pragma HLS PIPELINE
      m1_rs0.write(m1[j]);
    }
    m1[i] = m1_ws1.read();
  }
}

void process_unit(hls::stream<int> & m1_rs0, hls::stream<int> & m1_ws1) {
  int i;
  int mult;
  int j;
  outer: for(i = 0; i<10; i ++) {
#pragma HLS PIPELINE
    mult = 0;
    middle: for(j = 0; j<10; j ++) {
#pragma HLS PIPELINE
      mult = mult + m1_rs0.read();
    }
    m1_ws1.write( mult);
  }
}
void read_write(int * m1,) {
#pragma HLS INTERFACE s_axilite port=m1
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0 port=m1
  hls::stream<int> m1_rs0;
#pragma HLS STREAM variable=m1_rs0
  hls::stream<int> m1_ws1;
#pragma HLS STREAM variable=m1_ws1
#pragma HLS DATAFLOW
  fetch_unit0(m1, m1_rs0, m1_ws1);
  process_unit(m1_rs0, m1_ws1);
}
```

Listing 4.8: Solution for read after write of the same pointer

Units are created based upon the identified memory accessing element. A new IR that is not connected to the IR, that will become the execute unit, is created for each new access unit. An issue arises when the address of the memory accessing element depends on the result from another memory accessing element. Consider

```
1  void access_depend(int *m1, int *m2, int *out) {
2      int i, j, mult;
3      outer:for(i=0;i<10;i++) {
4          mult = 0;
5          middle:for(j=0;j<10;j++) {
6              mult = mult + m1[m2[j]];
7          }
8          out[i] = mult;
9      }
10 }
```

Listing 4.9: Memory access depending on another memory access

the code in Listing 4.9 the memory access *m1* depends on the memory access *m2*. Once a new IR is created for an access unit it also manipulates the node in the input IR so that the units become connected using a stream. The first accessing element is created for *m1* as it was the first one detected by Algorithm 1. Solving dependencies results in *m2* being accessed in the same access unit. It also replaces the memory accessing element (*m1[m2[j]]*) from the execute IR with a stream. The next memory accessing element *m2* results in another access unit, only this time *m2* doesn't exist in the execute IR. This results in an invalid code transformation as seen in Listing 4.10.

The solution to this issue is to allow for access units to be connected to other access units. So there will still be two access units one for *m1* and one for *m2* but the access unit for *m2* has its stream connected to the *m1* access unit instead of the execute unit. This results in the code shown in Listing 4.11.

It may be possible for loops and conditional statements to depend on memory accesses. Listing 4.12 shows a for loop that depends on memory accesses. This on its own doesn't introduce an issue. The issue arises when within that loop another memory accessing occurs. As every memory accessing element results in a new access unit, the loops and conditionals are copied from the execute unit to the access unit. This results in the memory access also being needed in the new access unit. Listing 4.13 shows how the memory accessing isn't connected from one access unit to another in the generated code. A potential solution, shown in Listing 4.14, is to this issue is to use additional streams for forwarding the accessed value from one unit to another.

```
1  void fetch_unit0(int * m1, int * m2, hls::stream<int> & m1_rs0) {
2    int i, j;
3    fetch_unit0_outer: for(i = 0; i<10; i ++) {
4  #pragma HLS PIPELINE
5      fetch_unit0_middle: for(j = 0; j<10; j ++) {
6  #pragma HLS PIPELINE
7        m1_rs0.write(m1[m2[j]]);
8      }
9    }
10 }
11 void fetch_unit1(int * m2, hls::stream<int> & m2_rs1) {
12   int i, j;
13   fetch_unit1_outer: for(i = 0; i<10; i ++) {
14 #pragma HLS PIPELINE
15     fetch_unit1_middle: for(j = 0; j<10; j ++) {
16 #pragma HLS PIPELINE
17       mult = mult * m1_rs0.read();
18     }
19   }
20 }
21 ...
22 void access_depend(int * m1, int * m2, int * out) {
23   ...
24   hls::stream<int> m1_rs0;
25 #pragma HLS STREAM variable=m1_rs0
26   hls::stream<int> m2_rs1;
27 #pragma HLS STREAM variable=m2_rs1
28   hls::stream<int> out_ws2;
29 #pragma HLS STREAM variable=out_ws2
30 #pragma HLS DATAFLOW
31   fetch_unit0(m1, m2, m1_rs0);
32   fetch_unit1(m2, m2_rs1);
33   process_unit(m1_rs0, m2_rs1, out_ws2);
34   write_unit2(out, out_ws2);
35 }
```

Listing 4.10: Invalid DAE code as a result of a memory access dependency issue

```
1  void fetch_unit0(int * m1, hls::stream<int> & m2_rs1, hls::stream<int> & m1_rs0) {
2    int i;
3    int j;
4    fetch_unit0_outer: for(i = 0; i<10; i ++) {
5  #pragma HLS PIPELINE
6      fetch_unit0_middle: for(j = 0; j<10; j ++) {
7  #pragma HLS PIPELINE
8        m1_rs0.write(m1[m2_rs1.read()]);
9      }
10   }
11 }
12 void fetch_unit1(int * m2, hls::stream<int> & m2_rs1) {
13   int i;
14   int j;
15   fetch_unit1_outer: for(i = 0; i<10; i ++) {
16 #pragma HLS PIPELINE
17     fetch_unit1_middle: for(j = 0; j<10; j ++) {
18 #pragma HLS PIPELINE
19       m2_rs1.write(m2[j]);
20     }
21   }
22 }
23 ...
24 void access_depend(int * m1, int * m2, int * out) {
25 ...
26   hls::stream<int> m1_rs0;
27 #pragma HLS STREAM variable=m1_rs0
28   hls::stream<int> m2_rs1;
29 #pragma HLS STREAM variable=m2_rs1
30   hls::stream<int> out_ws2;
31 #pragma HLS STREAM variable=out_ws2
32 #pragma HLS DATAFLOW
33   fetch_unit0(m1, m2_rs1, m1_rs0);
34   fetch_unit1(m2, m2_rs1);
35   process_unit(m1_rs0, out_ws2);
36   write_unit2(out, out_ws2);
37 }
```

Listing 4.11: Solution memory access depending on another memory access

```
void m_access(int *m1, int *m2, int *r, int *out) {
    int i, j;
    int mult;
    int r1;

    outer:for(i=0;i<10;i++) {
        mult = 0;
        r1 = r[i];
        middle:for(j=0;j<r1;j++) {
            mult = mult + m1[j] + m2[j];
        }
        out[i] = mult;
    }
}
```

Listing 4.12: Memory access depending on loops and conditionals that depend on memory accesses

```
void fetch_unit0(int * r, hls::stream<int> & r_rs0) {
  int i;
  fetch_unit0_outer: for(i = 0; i<10; i ++) {
#pragma HLS PIPELINE
    r_rs0.write(r[i]);
  }
}
void fetch_unit2(int * m1, hls::stream<int> & m1_rs1) {
  int i;
  int r1;
  int j;
  fetch_unit2_outer: for(i = 0; i<10; i ++) {
#pragma HLS PIPELINE
    fetch_unit2_middle: for(j = 0; j<r1; j ++) {
#pragma HLS PIPELINE
      m1_rs1.write(m1[j]);
    }
  }
}
void fetch_unit3(int * m2, hls::stream<int> & m2_rs2) {
  int i;
  int r1;
  int j;
  fetch_unit3_outer: for(i = 0; i<10; i ++) {
#pragma HLS PIPELINE
    fetch_unit3_middle: for(j = 0; j<r1; j ++) {
#pragma HLS PIPELINE
      m2_rs2.write(m2[j]);
```

```
    }
  }
}
void write_unit4(int * out, hls::stream<int> & out_ws3) {
  int i;
  write_unit4_outer: for(i = 0; i<10; i ++) {
#pragma HLS PIPELINE
    out[i] = out_ws3.read();
  }
}
void process_unit(hls::stream<int> & r_rs0, hls::stream<int> & r_rs1,
↪  hls::stream<int> & m1_rs1, hls::stream<int> & m2_rs2, hls::stream<int> &
↪  out_ws3) {
  int i, mult, r1, j;
  outer: for(i = 0; i<10; i ++) {
#pragma HLS PIPELINE
    mult = 0;
    r1 = r_rs0.read();
    middle: for(j = 0; j<r1; j ++) {
#pragma HLS PIPELINE
      mult = mult + m1_rs1.read() + m2_rs2.read();
    }
    out_ws3.write( mult);
  }
}
void m_access(int * r, int * r, int * m1, int * m2, int * out) {
...
  hls::stream<int> out_ws3;
#pragma HLS STREAM variable=out_ws3
#pragma HLS DATAFLOW
  fetch_unit0(r, r_rs0);
  fetch_unit2(m1, m1_rs1);
  fetch_unit3(m2, m2_rs2);
  process_unit(r_rs0, m1_rs1, m2_rs2, out_ws3);
  write_unit4(out, out_ws3);
}
```

Listing 4.13: Invalid DAE code resulting from loops that depend on access units

```
void fetch_unit0(int * r, hls::stream<int> & r_rs0) {
  int i;
  fetch_unit0_outer: for(i = 0; i<10; i ++) {
#pragma HLS PIPELINE
    r_rs0.write(r[i]);
  }
}

```

```
9  void fetch_unit2(int * m1, hls::stream<int> & m1_rs2, hls::stream<int> & r_rs0,
   ↪  hls::stream<int> & r_rs0_) {
10   int i;
11   int r1;
12   int j;
13   fetch_unit2_outer: for(i = 0; i<10; i ++) {
14 #pragma HLS PIPELINE
15     r1 = r_rs0.read();
16     r_rs0_.write(r1);
17     fetch_unit2_middle: for(j = 0; j<r1; j ++) {
18 #pragma HLS PIPELINE
19       m1_rs2.write(m1[j]);
20     }
21   }
22 }
23 void fetch_unit3(int * m2, hls::stream<int> & m2_rs3, hls::stream<int> & r_rs0,
   ↪  hls::stream<int> & r_rs0_) {
24   int i;
25   int r1;
26   int j;
27   fetch_unit3_outer: for(i = 0; i<10; i ++) {
28 #pragma HLS PIPELINE
29     r1 = r_rs0.read();
30     r_rs0_.write(r1);
31     fetch_unit3_middle: for(j = 0; j<r1; j ++) {
32 #pragma HLS PIPELINE
33       m2_rs3.write(m2[j]);
34     }
35   }
36 }
37 void write_unit4(int * out, hls::stream<int> & out_ws4) {
38   int i;
39   int mult;
40   int r1;
41   int r2;
42   int j;
43   write_unit4_outer: for(i = 0; i<10; i ++) {
44 #pragma HLS PIPELINE
45     out[i] = out_ws4.read();
46   }
47 }
48 void process_unit(hls::stream<int> & r_rs0, hls::stream<int> & m1_rs2,
   ↪  hls::stream<int> & m2_rs3, hls::stream<int> & out_ws4) {
49   int i;
50   int mult;
51   int r1;
52   int j;
```

```
53    outer: for(i = 0; i<10; i ++) {
54  #pragma HLS PIPELINE
55      mult = 0;
56      r1 = r_rs0.read();
57      middle: for(j = 0; j<r1; j ++) {
58  #pragma HLS PIPELINE
59        mult = mult + m1_rs2.read() + m2_rs3.read();
60      }
61      out_ws4.write( mult);
62    }
63  }
64  void m_access(int * r, int * r, int * m1, int * m2, int * out) {
65  #pragma HLS INTERFACE s_axilite port=r
66  #pragma HLS INTERFACE s_axilite port=m1
67  #pragma HLS INTERFACE s_axilite port=m2
68  #pragma HLS INTERFACE s_axilite port=out
69  #pragma HLS INTERFACE s_axilite port=return
70  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0 port=r
71  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem1 port=m1
72  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem2 port=m2
73  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem3 port=out
74    hls::stream<int> r_rs0;
75  #pragma HLS STREAM variable=r_rs0
76    hls::stream<int> r_rs1;
77  #pragma HLS STREAM variable=r_rs1
78    hls::stream<int> r_rs3;
79  #pragma HLS STREAM variable=r_rs3
80    hls::stream<int> m1_rs2;
81  #pragma HLS STREAM variable=m1_rs2
82    hls::stream<int> m2_rs3;
83  #pragma HLS STREAM variable=m2_rs3
84    hls::stream<int> out_ws4;
85  #pragma HLS STREAM variable=out_ws4
86  #pragma HLS DATAFLOW
87    fetch_unit0(r, r_rs0);
88    fetch_unit2(m1, m1_rs2, r_rs0, r_rs1);
89    fetch_unit3(m2, m2_rs3, r_rs1, r_rs3);
90    process_unit(r_rs3, m1_rs2, m2_rs3, out_ws4);
91    write_unit4(out, out_ws4);
92  }
```

Listing 4.14: Solution to memory access depending on loops that depend on memory accesses

The IR only contains the code for the selected target. As a result the implementation for function calls may not be available. Function call will also not be translated to the DAE architecture as they don't exist in the IR. Listing 4.15 shows code where a

memory accessing element has its pointer passed to another function that manipulates the memory element. There are two solutions possible, one would be to define the function call in another file and feed that to HLS, but this will not be transformed to the DAE architecture. Another is to move all the logic from a function call into the body of the targeted code as shown in Listing 4.16.

```
1  int calc(int *m1) {
2      int i;
3      int sum = 0;
4      middle:for(i=0;i<10;i++) {
5          sum = sum + m1[i];
6      }
7      return sum;
8  }
9
10 void m_function(int *m1, int *out) {
11     int i;
12     int sum;
13
14     outer:for(i=0;i<10;i++) {
15         sum = calc(m1);
16         out[i] = sum;
17     }
18 }
```

Listing 4.15: A function call from targeted code

```
1  void m_function(int *m1, int *out) {
2      int i;
3      int sum;
4
5      outer:for(i=0;i<10;i++) {
6          sum = 0;
7          middle:for(j=0;j<10;j++) {
8              sum = sum + m1[j];
9          }
10         out[i] = sum;
11     }
12 }
```

Listing 4.16: Solution to function calls from targeted code

To summarize the following limitations have been identified:

Limitation 1: Multiple access from the same memory pointer.

Limitation 2: Read after write from the same location in memory.

Limitation 3: A memory access depending on another memory access.

Limitation 4: A memory access that depends on branching and loops that depend on another memory access.

Limitation 5: Function calls (and its content) are not transformed to the DAE architecture.

Chapter 5

Implementation

This chapter describes how the framework is implemented. Section 5.1 describes how the C/C++ code is parsed into the IR. The framework implementation is described in Section 5.2. How the output of the framework is synthesized is described in Section 5.3. Lastly, how the framework is verified is described in Section 5.4.

5.1 Parsing

As discussed in previous chapters, the initial step is to get the C/C++ code in the IR that was presented in Section 4.6. For this the LLVM compiler infrastructure [19] is used. It is a widely used compiler infrastructure that consists of a frontend, an intermediate representation and a backend. A frontend is responsible for creating the LLVM IR. LLVM applies its own set of optimizations on top of the IR using multiple compiler passes. Finally, the backend uses the IR to generate the final machine code. See Section 2.4 for more information regarding compiler infrastructures.

While LLVM has its own intermediate representation, this does not fulfil the goal of this thesis: it is too abstract, information is lost from the higher-level algorithm. This is even true when working directly on the LLVM AST. As seen from the Section 2.4, preprocessing is applied on the tokens rather than after the generation of the AST. This means that macros may be missing from the AST. There is however an option to get the tokens while traversing the AST. For this reason, this thesis will use the AST to get structural information from the source code while storing the relevant tokens from every node from the AST. This ensures that no information is lost which is important to keep the generated code properly human readable.

LLVM provides two ways to interact with the AST: The libTooling API and the libclang. LibTooling is recommended for controlling the AST, while libclang allows for iterating over the AST. The main downside of using libTooling is its unstable API interface [20]. For this thesis the libclang API provides enough features as

there is no need to control the AST and the libclang API allows for extracting the structural information as well as the tokens that are relevant for a given part of the AST statement.

Figure 5.1 shows the final structure that is used to create the IR.

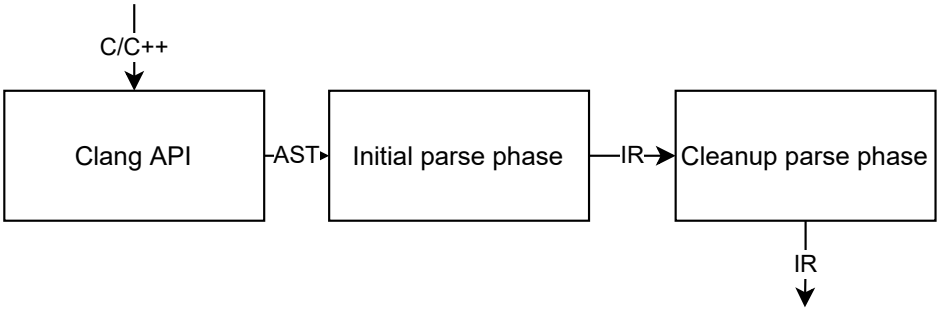


Figure 5.1: The parsing phases

Prior to parsing, the a code block is selected to be transformed to the DAE architecture. This means that the IR only contains the complete code that needs to be accelerated using the DAE architecture.

Libclang is used to parse the code into the intermediate representation. Using the libclang API the AST is traversed using what is called cursors (*CXCursor*). To get a cursor the input file needs to be parsed by LLVM, that can be done by using the *clang_indexSourceFile* function, it returns a translation unit which is the preprocessed source code that doesn't contain dependencies that could have been located in other source files.

The libclang function that allows for iteration over the AST is the *clang_visitChildren* function, which recursively iterates through the children. This function expects a cursor as its argument and calls a function repeatedly supplying different child cursors via function parameters. The *clang_visitChildren* function is used within that function to completely traverse the AST from beginning to the end.

As seen from Section 4.7 the AST cannot be directly parsed to the intermediate representation because it evaluates every statement which is not needed for the IR where structural information from loops and conditionals are enough. Instead, depending on what type of cursor is located it should either add a new node to the intermediate representation or parse the cursor and its children as a single statement into the intermediate representation. Types that would result in a node that is located at an *in* edge are: loops and conditional statements. All others are parsed into a node located on the *next* edge of the previous node. Instead of using parsing every node the *clang_tokenize* function is used to get the string representation directly from the source code, this is to avoid having preprocessed tokens within the

output of the framework, this way a macro definition remains in the output instead of being replaced by what the preprocessor has processed (for example a literal number).

As the structure of the LLVM AST differs from the structure defined by the IR another internal structure is used that stores all nodes from the AST in subsequent nodes. So a node only has a *previous* and *next* node. To ensure that this internal structure can be transformed into the final IR the internal structure has an additional field(*offset*) that stores to what extend the nodes is nested in the entire intermediate representation.

Figure 5.2 shows an overview of the initial parsing phase. It consists of traversing the the AST by recursively calling the *clang_visitChildren* function on every node. Depending on the type of statement (loop, conditional, compound, the rest) a new node is appended into a long list in known as the *Internal Parsing Structure*. This list is transformed into the IR.

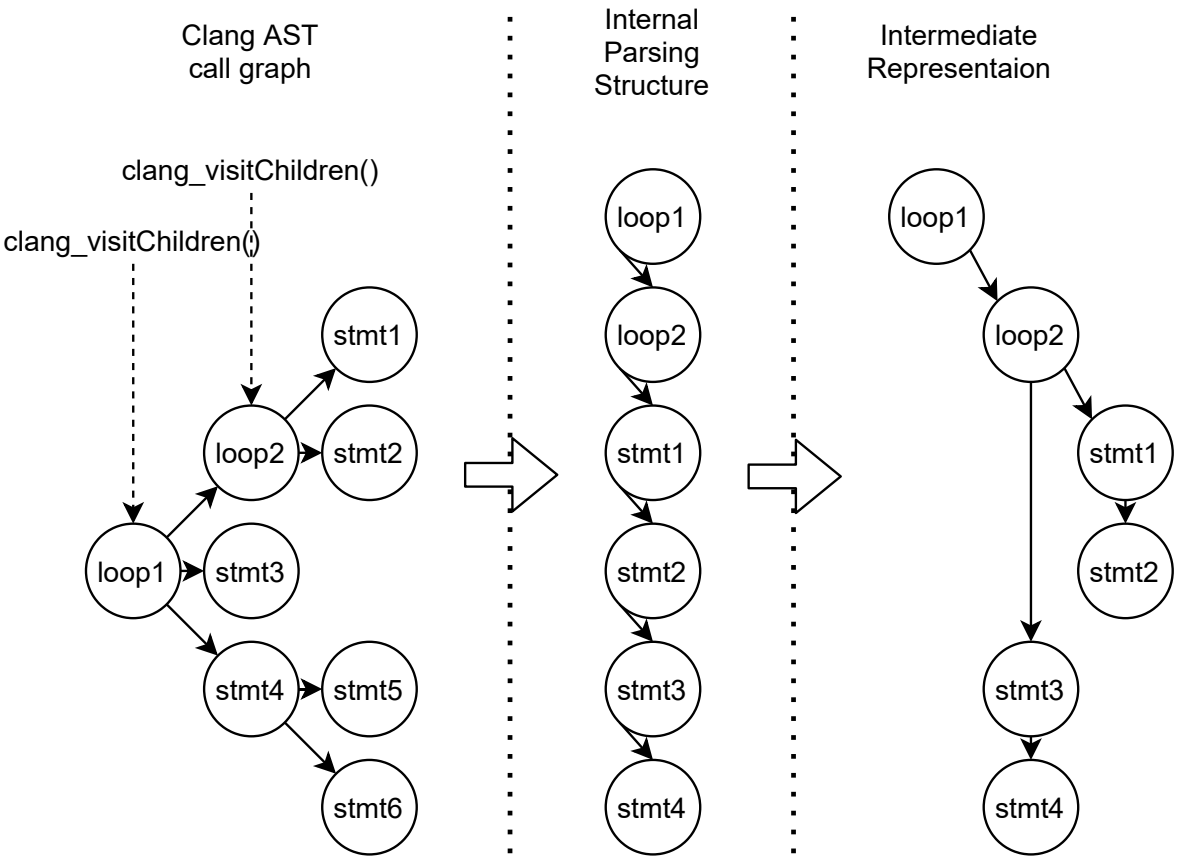


Figure 5.2: The initial parsing phase

During the parsing of the LLVM AST to the IR there are some steps that introduce more data in the structure that is not needed as it is already there by means of the *next* and *in* edges. An example of this is the *CompoundStmt*. Whenever curly

brackets are used in the code it introduces a *CompoundStmt* which is another child in the LLVM AST. In the LLVM AST a loop or a conditional statement has a single statement as its body, using a *CompoundStmt* introduces the ability of using multiple statements within the body of a loop or a conditional statement. The IR uses an *in* edge to determine whether a statement is part of the body of another statement, there is no need to store a *CompoundStmt*. The internal structure does base the offset on the number at which a statement is nested, a *CompoundStmt* increases this offset. Part of the parsing phase is to clean this up after the complete LLVM AST has been parsed.

Another important part of this phase is to handle the if-statement. The *clang_tokenize* function relies on the location range in the source code. This location range is obtained from the cursor that is used to traverse the AST. Listing 5.1 shows a simple if-else statement and Figure 5.3 shows the corresponding LLVM AST. When the cursor is positioned at the *IfStmt* the *clang_tokenize* function can be used to get all the tokens related to the if-statement and its expression. However, notice that there is no statement that represents the *else* part, there is only the body of the else statement. That is because the LLVM AST relies on the number of children to determine if the if-statement has an else part. As there is no cursor for the *else* part (only the body) there is no source range that can be used to get the corresponding tokens for it. This means that while cleaning up the IR a new node is added before the contents of the else statement.

```
1  if (r) {
2      r += c;
3  } else {
4      r -= c;
5  }
```

Listing 5.1: If-else statement

```
IfStmt 0x55ff148684b8 <line:4:1, line:8:1> has_else
|-ImplicitCastExpr 0x55ff14868360 <line:4:5> 'bool' <IntegralToBoolean>
| `-ImplicitCastExpr 0x55ff14868348 <col:5> 'int' <LValueToRValue>
|   `-DeclRefExpr 0x55ff14868328 <col:5> 'int' lvalue Var 0x55ff14868210 'r' 'int'
|-CompoundStmt 0x55ff14868400 <col:8, line:6:1>
| `-CompoundAssignOperator 0x55ff148683d0 <line:5:5, col:10> 'int' lvalue '+=' ComputeLHSTy='int' ComputeResultTy='int'
|   |-DeclRefExpr 0x55ff14868378 <col:5> 'int' lvalue Var 0x55ff14868210 'r' 'int'
|   `-ImplicitCastExpr 0x55ff148683b8 <col:10> 'int' <LValueToRValue>
|     `-DeclRefExpr 0x55ff14868398 <col:10> 'int' lvalue Var 0x55ff14868290 'c' 'int'
`-CompoundStmt 0x55ff148684a0 <line:6:8, line:8:1>
  `-CompoundAssignOperator 0x55ff14868470 <line:7:5, col:10> 'int' lvalue '-=' ComputeLHSTy='int' ComputeResultTy='int'
    |-DeclRefExpr 0x55ff14868418 <col:5> 'int' lvalue Var 0x55ff14868210 'r' 'int'
    `-ImplicitCastExpr 0x55ff14868458 <col:10> 'int' <LValueToRValue>
      `-DeclRefExpr 0x55ff14868438 <col:10> 'int' lvalue Var 0x55ff14868290 'c' 'int'
```

Figure 5.3: if-else statement AST

5.2 Framework implementation

The framework implements the algorithms described in the Chapter 4.

The first step after the creation of the IR is the identification of the memory accessing elements. Algorithm 1 specifically searches the intermediate representation for a variable with an opening and a closing bracket. This is identified as the memory accessing variable. libclang offers the ability to get the cursor where the current cursor refers to using the *clang_getCursorReferenced* function. For example *m1[i]* refers to a definition *type *m1*. If the referred cursor is located in the function parameter or above it (a global variable) and it has the form of an array then it is added to the list of memory accessing elements. libclang has a special type for when a cursor is located in the function parameter list: *CXCursor_ParmDecl*. Function parameters are added to the list of memory accessing elements if it matches the form of *type * var* or *type var[x]*. This method is slightly different from the algorithm defined by Algorithm 1 but achieves a similar result but more limiting as local arrays are not used, they stay local in the execute unit.

For every single element in the list of memory accessing elements, the IR is traversed from the root. At this point it uses Algorithm 1 to find every memory access that uses this memory accessing element. Assume a recursive function *DAE_LocateAndHandleUniqueMemoryParams* that is called for every node. When it finds a token that matches the name of the element from the list of memory accessing elements then the next token should start with an open square bracket (*[*), everything after that is used for address generation until the corresponding close square bracket (*]*) is found. At this location a reverse copy of the IR is created, this will become an access unit. Using Algorithm 4 it is determined if the access is a read or write. The exact memory accessing element was already defined (For example *m1[i+j]*), if this is located at the start of the statement then it is an access unit that writes data to memory. Otherwise it results in an access unit that reads data from memory. The entire statement is replaced by a read or write from memory to a stream, so when a write was detected the statement is replaced by a read from stream and write to memory (Example: *m1[i] = stream.read()*) when a read was detected it is replaced by the corresponding read from memory into a stream (Example: *stream* $<<$ *m1[i]*). The same statement is also replaced in the original IR with a read or write from a stream. Token manipulation is used to ensure that the contents that are written remain the same. Meaning that if a write is found it replaces the contents up to the equals sign with the stream write function (i.e. *m1[i] = 42* transforms into *stream* $<<$ *42*).

Xilinx provides a stream library that is used to connect the different units. It is part of the HLS tool that they provide. The stream is implemented as a FIFO using

the Xilinx FIFO Generator.

At this point all the IRs are created for the access and execute units. Next, the function definitions are added before the root nodes of every IR. Access units have a memory accessing element pointer and a stream, followed by other function parameters. During the parsing phase any function parameter that wasn't a memory accessing element was added to a separate list of data dependencies. The same is true for other locally defined variables. A locally defined variable is detected by using information about where a definition of a variable is located in the code with relation to the function header and the start of the IR. These steps are required as the IR only contains the target to be transformed to the DAE architecture. The function header of the execute unit does never contain a memory accessing element, instead it only has streams and other function parameters (primitive variables).

Loop pipelining is a method to reduce the initiation interval at the cost of an increased area usage. Loop pipelining is enabled for a loop by adding a *HLS PIPELINE* pragma as the first child of a loop node. Algorithm 5 shows algorithmically how loop pipelining is enabled for all nodes in the IR.

Algorithm 5 Loop pipelining

Precondition: $node$ Node located in the IR.

```
1: function PIPELINE_LOOPS(node)
2:     if node is a loop node then
3:         node ← Add a pragma node on the in edge
4:     end if
5:     if node has in edge then
6:         pipeline_loops(in node)
7:     end if
8:     if node has next edge then
9:         pipeline_loops(next node)
10:    end if
11: end function
```

Local arrays can be optimized for speed by using array partitioning, creating more ports for accessing the array. During parsing local variables are detected by using information from where it is defined and if it is position in the function that has the selected target. Local arrays are found if they match the form: *type var[x]*.

The core function, the function that is the root of the selected target, is the last function that is created. It is the connecting glue that defines the streams and connects the units. The function only has memory accessing elements and the function parameters as the function parameter. These are the ports that will become visi-

ble after HLS. As mentioned in Section 4.5 AXI4 interfaces are used for the ports. The *HLS INTERFACE m_axi port=x* pragma is added for every memory accessing element to configure the HLS tool to use the AXI4 interface. Then every stream is defined using *hls::stream<Type> stream_name*.

The most important part before calling every unit is to add the dataflow directive: *HLS DATAFLOW*. This schedules all units to run in parallel. Lastly, all units are called from the core function. The order at which they are added is not relevant as they run in parallel.

As the code is currently in the IR it needs to be translated to source files that the HLS tool can use. The implemented method to do this is to iterate over the IR and write all tokens as a new line to a source file. Generally a token is surrounded by a space, with the exception of a dot and sometimes also brackets (*<>, [], (), and {}*).

Appendix A shows the translated C/C++ code shown in Listing 4.1.

5.3 Hardware synthesis

The framework targets the Xilinx Vitis software platform [21] for the hardware synthesis phase. This platform is an umbrella project that encompasses the complete flow to generate and test C/C++ source code on an FPGA. A host program is responsible for loading a kernel, the DAE C/C++ code generated from the framework, onto the FPGA and setting it up. The kernel is generated using HLS or a hardware description language directly. This kernel is automatically synthesized, placed and routed for a targeted FPGA device.

The Vitis platform has the ability to perform hardware and software simulation using the same host code. This allows the engineer to use the same host code and kernel code for software and hardware emulation reducing the probability for introducing errors in verification code because there is no need for duplicated verification code.

The Vitis platform uses the Vitis HLS [22] tool to synthesize the C/C++ algorithm to a hardware description language. This HDL is then used in the Vivado Design Suite to be synthesized to a netlist, which is in turn used for the place and route stage, that is responsible for making sure that the netlist actually will fit on a target FPGA using the constraints imposed by it.

Xilinx Runtime Library (XRT) [23] is used to facilitate communication between a host program and a hardware accelerator, it defines a standard API that is similar to OpenCL for easy development. It consists of a set of user space libraries and Linux kernel drivers. This allows the host program to download a kernel onto the FPGA using XRT while also managing the data for both. The host program is responsible for starting the kernel, for which XRT also provides the necessary APIs.

5.4 Verification

One of the initial steps of the framework is to parse the C/C++ code into the intermediate representation defined in Section 4.6. A key part of the IR is that the original C/C++ code can be regenerated using the framework. Meaning that correct parsing and writing generation can be verified by rewriting the input via the IR.

The generated C/C++ code by the framework alters the architecture of the code. To ensure that this doesn't cause for erroneous behaviour it is verified using the Xilinx simulation capabilities.

The following chapter will go into detail on how the framework is evaluated.

Chapter 6

Evaluation and Discussion

This chapter shows the results of the achievable speedup and area usage and power usage of the MachSuite benchmark [8]. The results are compared against a baseline implementation that will be explained in the following section.

6.1 Experimental setup

The Berkeley Dwarfs benchmark [24] provides a set of dwarfs. A dwarf is an algorithmic method that describes a specific computation such as dense linear algebra and structured grids. The OpenDwarfs benchmark [7] provides a set of algorithms and implementations of these algorithmic methods in the OpenCL programming language.

The MachSuite benchmark [8] provides a different set of algorithms and implementations of these are written in C/C++. It specifically targets HLS, by providing implementations that can be synthesized directly using the Xilinx HLS tooling. The framework processes source code written in the C/C++ programming language, for this reason the MachSuite benchmark is used.

As described in Section 4.8 not all memory accessing structures are supported by the framework. Table 6.1 shows the various limitations that are identified for any of the implemented algorithms provided by the MachSuite benchmark. For this reason a subset of the 19 defined benchmarks is used for the evaluation of the framework. Table 6.2 shows the benchmarks that will be used for evaluation.

In this work, there are three different versions considered:

- *Baseline*: The accelerator as provided by the MachSuite benchmark with minimal changes so that the interfacing between it and the subsequent other version have an identical interface. This ensures a fair comparison amongst the different versions.
- *Framework*: The DAE accelerator generated by the framework from this thesis.

Benchmark	Limitation as specified in Section 4.8
aes/aes	Limitation 1 & Limitation 2 & Limitation 5
backprop/backprop	Limitation 1 & Limitation 2 & Limitation 5
bfs/bulk	Limitation 1 & Limitation 2 & Limitation 4
bfs/queue	Limitation 1 & Limitation 2 & Limitation 4
fft/strided	Limitation 1 & Limitation 2 & Limitation 4
fft/transpose	Limitation 1 & Limitation 2 & Limitation 5
gemm/ncubed	None
gemm/blocked	Limitation 2
kmp/kmp	Limitation 1 & Limitation 4
md/knn	Limitation 1
md/grid	Limitation 1 & Limitation 2 & Limitation 4
nw/nw	Limitation 1 & Limitation 2
sort/merge	Limitation 1 & Limitation 2 & Limitation 5
sort/radix	Limitation 1 & Limitation 2 & Limitation 5
spmv/crs	Limitation 3 & Limitation 4
spmv/ellpack	Limitation 3
stencil/stencil2d	None
stencil/stencil3d	Limitation 1
viterbi/viterbi	Limitation 1

Table 6.1: Limitations from the framework imposed on the benchmarks

- *Optimized*: Further manual optimizations applied on top of the DAE accelerator generated by the framework presented in this thesis.

The *optimized* version has a large domain of exploration so it may be omitted for some benchmarks as at the time of writing this report, no further optimizations were found that didn't drastically change the entire structure of the source code which falls outside of the scope of this thesis.

The framework uses the Xilinx Vitis 2021.1 platform for the high-level synthesis phase. Vitis is a software platform that uses the Vitis HLS tool for high-level synthesis [22]. The Xilinx Alveo U200 Data Center accelerator card is used as the target FPGA. This card is connected to a host CPU using a PCIe interface.

Again, functional correctness of the kernel is verified for the software implementation as well as the hardware implementation. For this purpose, the same host code is used to ensure that the host code doesn't contain erroneous behaviour.

Benchmark	Description
gemm	Matrix multiplication $O(N^3)$
spmv	Sparse matrix vector multiplication
stencil2d	Two-dimensional stencil computation

Table 6.2: Benchmarks considered

6.2 gemm benchmark

General Matrix multiplication of the form $prod = m1 * m2$ produces an output matrix $prod$ based on the multiplication of matrix $m1$ and $m2$. The benchmark uses a naive $O(N^3)$ implementation.

Type	Execution time (ms)
Baseline	1.426
Framework	1.420

Table 6.3: gemm: Kernel execution time

Table 6.3 shows the kernel execution time based on profiling the kernel using hardware emulation. It shows only a small increase in speedup (1.004x). Table 6.4

shows the latency and initiation interval based on the HLS tool (Vitis HLS) report. A similarly small decrease in initialization interval is also visible.

Type	Latency (cycles)	Initiation Interval (cycles)
Baseline	262815	262816
Framework	262746	262612

Table 6.4: gemm: Delay and initiation interval

Type	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Baseline	64 (2.96)	11 (0.16)	26244 (1.11)	14692 (1.24)
Framework	64 (2.96)	11 (0.16)	26514 (1.12)	15853 (1.34)

Table 6.5: gemm: Kernel area usage

Table 6.5 shows that the kernel area has slightly increased. The baseline and framework code is slightly different this alone could cause a slightly increased area usage because place and routing differs depending on the code.

Type	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Baseline	296 (13.7)	18 (0.26)	275264 (11.64)	188827 (15.97)
Framework	296 (13.7)	18 (0.26)	275472 (11.65)	188646 (15.96)

Table 6.6: gemm: Total chip area usage

The total chip area used is shown in Table 6.6. It shows that while the BRAM and the number of DSPs doesn't change, the number of flip-flops (FFs) and LUTs show a relatively small change. The total power consumption of the FPGA, see Table 6.7, also only shows a small decrease in power usage when moving towards the DAE architecture. The miscellaneous column represents a part of the power usage that stays mostly constant. It consists of the URAM, PLL, MMCM, I/O and GTY power usage. The GTY transceivers take up the majority of the power usage (4.360 Watts). The 0.004 Watts difference comes from the URAM, all others stay constant.

	Total	Hard IP	Dynamic						Static
			Clocks	Signals	Logic	BRAM	DSP	Misc	
Baseline	15.491	0.391	1.935	1.909	1.124	1.683	0.025	5.690	2.733
Framework	15.340	0.391	1.985	1.704	1.124	1.694	0.017	5.694	2.731

Table 6.7: gemm: Power usage (Watts)

6.3 spmv benchmark

The Sparse Matrix Vector multiplication benchmark calculates the result of a sparse matrix multiplied with a vector. The matrix is stored in the standardized Compressed Sparse Row (CRS) format [25]. Only nonzero values of the matrix are stored along with the position of this value. The CRS format uses a specific encoding method for storing the position. This leads to memory accessing patterns where a data access depends on the result of another data access.

As described in Table 6.1 the spmv benchmark cannot be used directly with the HLS tool as the framework has limitations that result in invalid code for the HLS tool. This benchmark uses memory accesses that depend on other memory accesses (Limitation 3). The solution proposed in Listing 4.11 is implemented for the spmv benchmark. The CRS encoding introduces for-loops where the range is not constant, it depends on data supplied from a memory accessing element. This leads to a limitation where loops in access units depend on data from other access units (Limitation 4). Here the solution proposed Listing 4.14 is implemented. See the code in Appendix B.2 for the exact implementation.

The related work by Charitopoulos et al. [2] also tests against this benchmark. Contrary to the other benchmarks, for this benchmark they provide a schematic overview of how this benchmark is implemented. See Figure 6.1 for an adapted overview of the benchmark. All blocks are accessing units with the exception of the *Memory* and the *Process* block. It describes dedicated queues for requesting data from memory and dedicated queues for forwarding the data into other units.

The schematic in Figure 6.1 introduces a further separation between address generation and memory access, it uses two queues one for each. The address generation and memory accessing occur in parallel (there is a separate task that calculates the addresses and a separate task that sends the memory accessing requests). Another difference is that each memory accessing element does not have a dedicated access unit. Instead they are combined to follow the program data

flow.

Figure 6.1 shows that the *row_delimiters* are transferred to most units with the exception being the *out* access unit. Different streams are used for every unit that needs row delimiters this is needed as a stream can only have a single producer and consumer. This means that when *col* reads a row delimiter from the row delimiter access unit it also needs to forward this into a different stream that is connected to the next unit. The forwarding of the row delimiters and data accessing don't actually depend on each other allowing for parallelization. The automatically generated code from the framework has these code changes manually applied.

The optimized implementation represents the implementation created by Charitopoulos et al. [2].

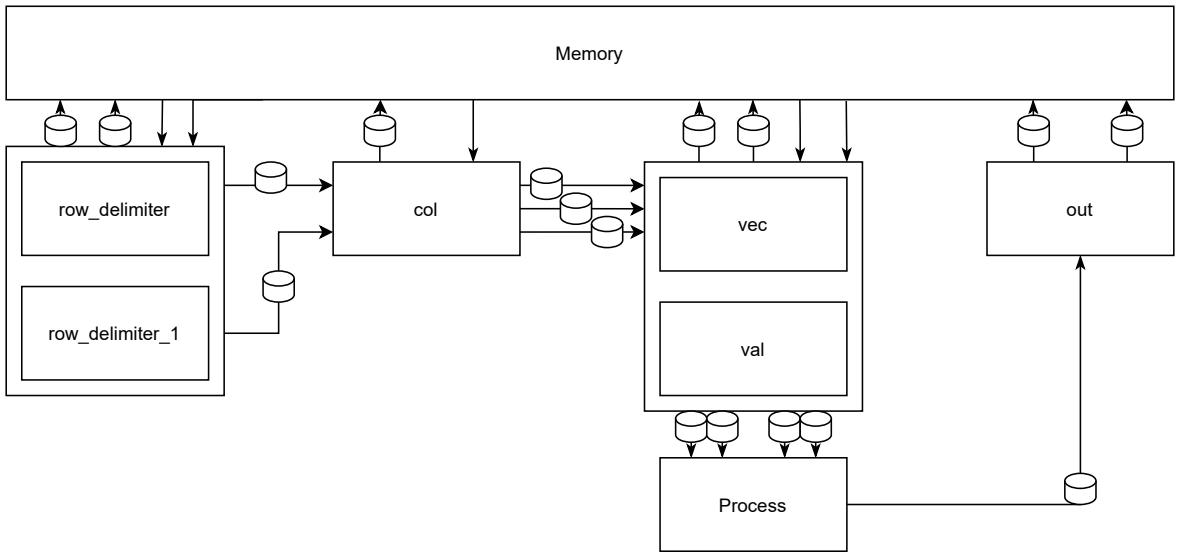


Figure 6.1: spmv benchmark schematic adapted from [2]

See Table 6.8 for an overview of achieved execution times as observed from the profiler. The move towards the DAE architecture has a large improvement over the baseline implementation (1.63x). But an even higher speedup is observed by the optimized implementation (2.94x) when compared to the baseline implementation. The speedup is gained by letting the row delimiters run in parallel with the data accesses.

For this benchmark it is not possible to get the latency and the initiation interval from the HLS tool (Vitis HLS) report as the loop iterations within this benchmark depend on the data provided by the host application. The number of iterations are not constant during the synthesis phase.

An overall increase in flip-flop and LUTs can also be seen in the Table 6.9.

Type	Execution time (ms)
Baseline	0.429
Framework	0.264
Optimized	0.146

Table 6.8: spmv: Kernel execution time

Type	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Baseline	20 (0.93)	11 (0.16)	5774 (0.24)	7198 (0.61)
Framework	24 (1.11)	11 (0.16)	8330 (0.35)	10914 (0.92)
Optimized	16 (0.74)	11 (0.16)	10935 (0.46)	10959 (0.93)

Table 6.9: spmv: Kernel area usage

Type	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Baseline	321 (14.9)	18 (0.26)	256458 (10.85)	179867 (15.21)
Framework	341 (15.8)	18 (0.26)	261850 (11.07)	183572 (15.53)
Optimized	320 (14.8)	18 (0.26)	260160 (11.00)	182610 (15.45)

Table 6.10: spmv: Total chip area usage

Table 6.10 shows the total chip area usage. It shows a relatively small increase in BRAM usage when using the framework generated output, while it decreases in the optimized version. The number of DSPs stays constant, which is also reflected by the kernel area usage in Table 6.9. The number of flip-flops and Block RAM (BRAM) also seems to follow the same figure as the number of BRAM.

It appears that the same is true for the power figure in Table 6.11. Overall the power usage is increased with the framework, but then reduced again in the optimized one due to the changes applied on top of the framework version. The miscellaneous column represents a part of the power usage that stays mostly constant. It consists of the URAM, PLL, MMCM, I/O and GTY power usage. The GTY transceivers take up the majority of the power usage (4.360 Watts). The 0.003 Watts difference comes from the I/O, all others stay constant.

	Total	Hard IP	Dynamic						Static
			Clocks	Signals	Logic	BRAM	DSP	Misc	
Baseline	15.093	0.391	1.915	1.508	1.091	1.729	0.025	5.708	2.726
Framework	15.362	0.391	1.984	1.521	1.119	1.881	0.027	5.708	2.732
Optimized	15.129	0.391	1.963	1.505	1.102	1.703	0.028	5.711	2.727

Table 6.11: spmv: Total power usage (Watts)

During synthesis of the baseline implementation the HLS tool reported issues shown in Figure 6.2. These issues were not reported for the output code from the framework and the manually optimized version of the code. The *Could not analyze pattern* error shows up when the logic to access data from an *m_axi* interface does not allow for burst transfers is too complex. Xilinx does not specify the logic is too complex [26].

BURSTS AND WIDENING MISSED

HW Interface	Variable	Loop	Problem
m_axi_gmem3	vec	spmv_2	Could not analyze pattern
m_axi_gmem2	rowDelimiters	spmv_1	Stride is incompatible
m_axi_gmem0	val	spmv_1	Access call is in the conditional branch
m_axi_gmem1	cols	spmv_1	Access call is in the conditional branch
m_axi_gmem0	val	spmv_2	Could not widen since Type double size is greater than or equal to alignment 8(bytes)
m_axi_gmem1	cols	spmv_2	Could not widen since Type i32 size is greater than or equal to alignment 4(bytes)
m_axi_gmem2	rowDelimiters		Could not widen since Type i32 size is greater than or equal to alignment 4(bytes)
m_axi_gmem0	val	spmv_2	Could not widen since Type double size is greater than or equal to alignment 8(bytes)
m_axi_gmem1	cols	spmv_2	Could not widen since Type i32 size is greater than or equal to alignment 4(bytes)
m_axi_gmem2	rowDelimiters		Could not widen since Type i32 size is greater than or equal to alignment 4(bytes)

Figure 6.2: High-level synthesis too complex to optimize baseline spmv

6.4 stencil2d benchmark

This benchmark consists of a two-dimensional stencil computation using a 9-point square stencil. The 9-point square stencil is a fixed size filter that is applied on top of the two-dimensional matrix.

The kernel execution times from hardware emulation profiling are shown in Table 6.12, it shows a speedup of 1.23x. Table 6.13 shows the latency and initiation interval of the kernel based on the report provided by the HLS tool (Vitis HLS). A reduction in latency and initiation interval is visible.

Type	Execution time (ms)
Baseline	0.303
Framework	0.246

Table 6.12: stencil2d: Kernel execution times

Type	Latency (cycles)	Initiation Interval (cycles)
Baseline	88957	88958
Framework	70525	70382

Table 6.13: stencil2d: Delay and initiation interval

From the kernel area usage in Table 6.14 a reduction in the number DSPs is visible while the number of FFs and LUTs is increased.

Type	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Baseline	8 (0.37)	21 (0.31)	4428 (0.19)	4055 (0.34)
Framework	8 (0.37)	9 (0.13)	4888 (0.21)	4605 (0.39)

Table 6.14: stencil2d: Kernel area usage

The total chip usage shown in Table 6.15 reflects the same change when compared to the kernel area usage albeit at a much smaller difference.

Type	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Baseline	281 (13.0)	28 (0.41)	246734 (10.44)	173639 (14.69)
Framework	281 (13.0)	16 (0.23)	247953 (10.49)	174901 (14.79)

Table 6.15: stencil2d: Total chip area usage

Table 6.16 shows the power usage figure for the baseline and the framework. It shows an increase in power usage for all elements with the exception of the DSPs but the number of DSPs is also reduced in the framework version when looking at

Table 6.15. The miscellaneous column in Table 6.16 represents a part of the power usage that stays mostly constant. It consists of the URAM, PLL, MMCM, I/O and GTY power usage. The GTY transceivers take up the majority of the power usage (4.360 Watts).

	Total	Hard IP	Dynamic						Static
			Clocks	Signals	Logic	BRAM	DSP	Misc	
Baseline	14.349	0.391	1.790	1.348	0.975	1.382	0.042	5.708	2.713
Framework	14.453	0.391	1.853	1.401	0.985	1.386	0.014	5.709	2.715

Table 6.16: stencil2d: Total power usage (Watts)

For this benchmark issues were also reported by the HLS tool shown in Figure 6.3. Here too was the *Could not analyze pattern* issue reported, indicating a too complex logic failing to enable burst transfers on the *m_axi* interface.

BURSTS AND WIDENING MISSED

HW Interface	Variable	Loop	Problem
m_axi_gmem0	orig	stencil_label3	Stride is incompatible
m_axi_gmem2	filter	stencil_label2	Could not analyze pattern
m_axi_gmem1	sol	stencil_label1	Stride is incompatible
m_axi_gmem2	filter	stencil_label4	Sequential access length is not divisible by 2
m_axi_gmem2	filter	stencil_label4	Start index of the access is unaligned
m_axi_gmem0	orig	stencil_label4	Could not widen since Type i32 size is greater than or equal to alignment 4(bytes)
m_axi_gmem2	filter	stencil_label4	Sequential access length is not divisible by 2
m_axi_gmem2	filter	stencil_label4	Start index of the access is unaligned
m_axi_gmem0	orig	stencil_label4	Could not widen since Type i32 size is greater than or equal to alignment 4(bytes)

Figure 6.3: High-level synthesis too complex to optimize baseline stencil2d

6.5 Summary

Figure 6.4 summarizes the overall speedup achieved when comparing the output of the framework against the baseline. Overall, a small increase in speedup is observed, with the gemm benchmark having such a low speedup, at which point the speedup could be just a profiling accuracy offset. Table 6.8 shows that due to the human readable feature further improvements in speedup are still possible.

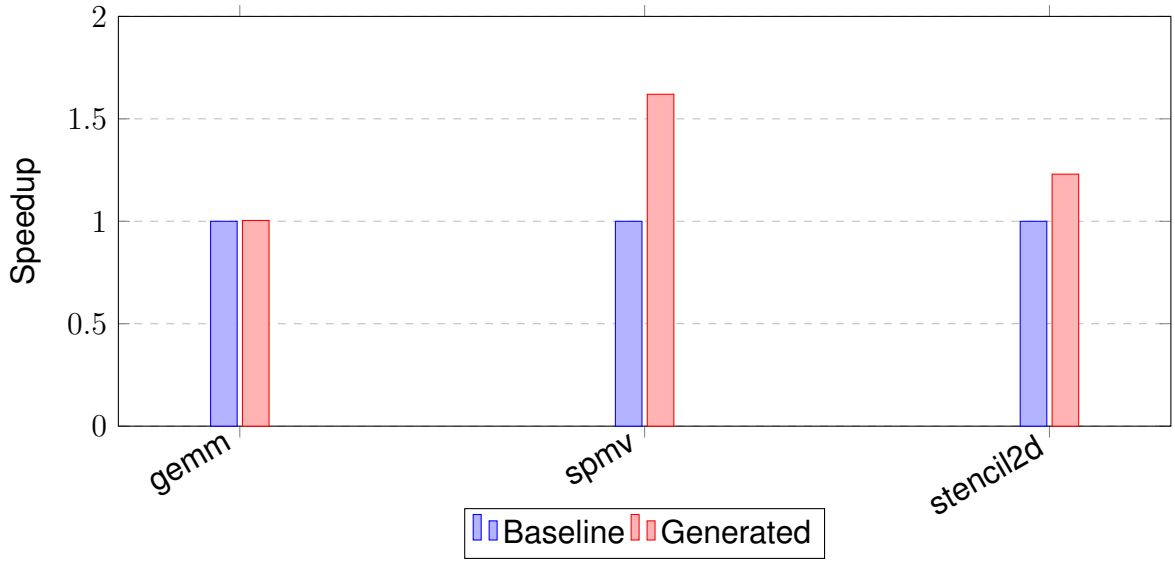


Figure 6.4: Speedup comparison with all benchmarks

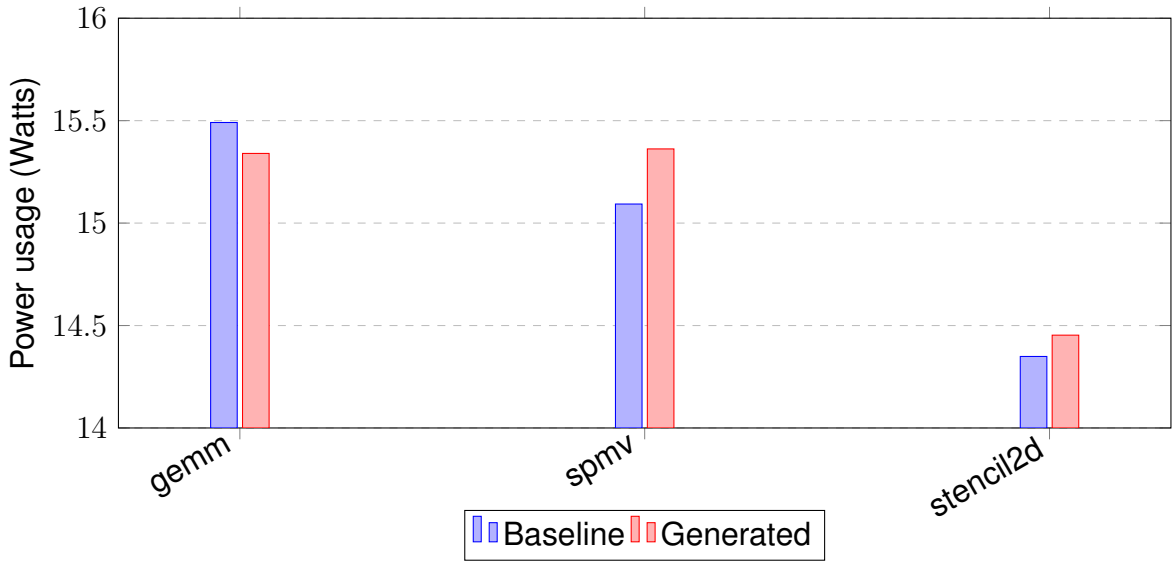


Figure 6.5: Power usage comparison with all benchmarks

The power figure seems to be somewhat similar in Figure 6.5. The framework seems to consume slightly less power for the gemm benchmark, but the speedup for this benchmark is also non-existent. The spmv and stencil2d have a larger speedup, this seems to result into a higher power usage. The power differences among the benchmarks is relatively low: -0.98% (gemm), +1.78% (spmv), +0.72% (stencil2d).

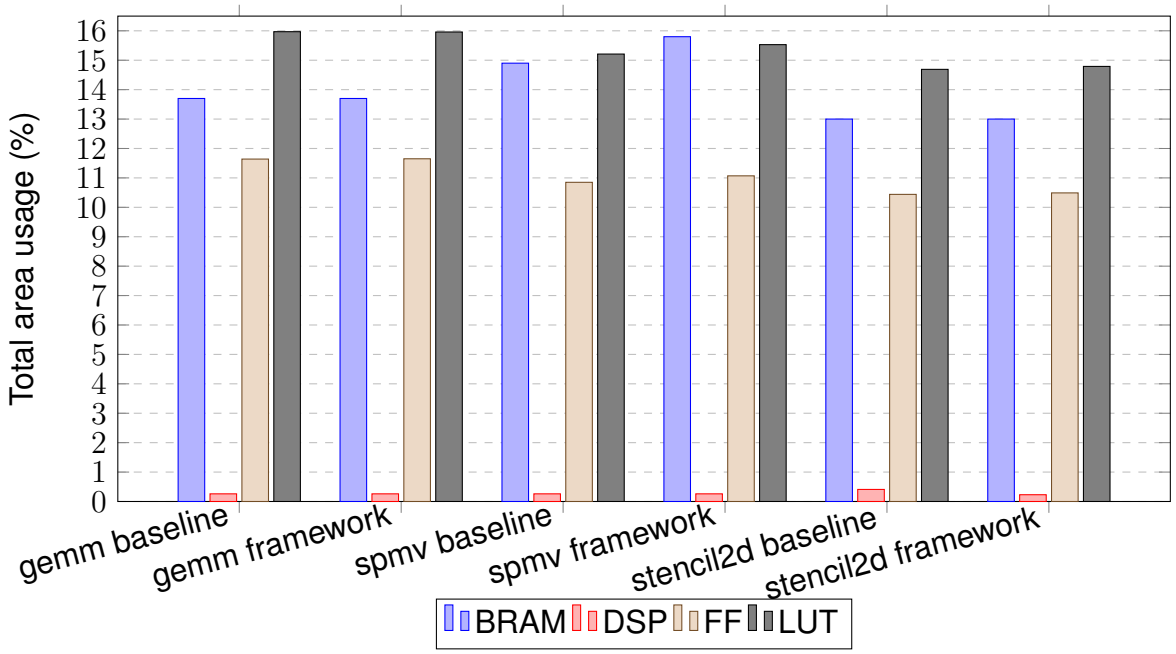


Figure 6.6: Total chip area usage with all benchmarks

The total chip area usage in Figure 6.6 shows a small increase in BRAM, FFs and LUTs for all benchmarks. The number of DSPs appears to stay constant for the gemm and spmv benchmarks while it decreases for the stencil2d benchmark. Notice that the differences are small due to the kernel area usage also being small in comparison to the size of the target FPGA.

The spmv and stencil2d benchmark showed an issue during high-level synthesis where it could not enable burst transfers for some memory accessing elements for the baseline version. After transformation to the DAE architecture this issue is solved. This issue does not occur with the gemm benchmark.

Chapter 7

Conclusions and recommendations

7.1 Conclusion

This thesis presented a framework that automatically translates C/C++ code to the DAE architecture while keeping the DAE architecture readable to the engineer.

One of the goals of this thesis is that the framework automatically generates C/C++ code in the DAE architecture that is human readable. The implemented method allows an engineer to apply further manual optimizations targeted for the HLS tool for specific FPGAs.

The result of the framework is validated against the MachSuite benchmark [8]. Taking the baseline implementation provided by the benchmark and transforming that automatically to the DAE architecture and inserting directives to optimize for speed. This resulted in a maximum increase in the speedup of 1.63x, it is however highly dependent on the structure of the algorithm.

Automatically transforming the baseline version to the DAE architecture results in data accessing that is less complex as it only contains the logic for a memory accessing element and stream logic. Other, computational, logic remains in the execute unit. The reduces complexity for data accessing results in the HLS tool being able to better optimize the memory data access which in turn improves the speedup of an algorithm.

The next chapter will answer the research questions that were formulated for this thesis.

7.1.1 Research questions

The first three research sub-questions directly relate to the DAE architecture. A key challenge for generating the DAE architecture is the identification of memory accessing elements in the source code. This immediately leads to the following research sub-question:

How can one extract memory and computational parts from the C/C++ code?

The source code is first parsed to the intermediate representation defined in the Section 4. Further analysis and manipulation is using this IR. This intermediate representation is considered to be the computational part. Memory accessing parts are replaced by corresponding queues that connect to the dedicated memory access units.

By analyzing the specific ways that memory is accessed in a programming language, in this case C/C++, a pattern is observed where memory is accessed by using the index of an array. Section 4.3 describes this accessing method in detail while also describing other memory accessing patterns. The index of the array is used for the address generation information of the memory access. The actual value of the address generated can have dependencies elsewhere in the code. As the memory accessing logic is moved to a dedicated accessing unit so does the address generation logic. This results into the following research sub-question:

How can one solve dependencies (data access) within the different access and execute units?

A memory access has an address associated to it, an address may be constructed from different places. Taking the dependencies from the memory access address, the code is reverse traversed until it has found every data dependency. As the code is in the intermediate representation, reverse traversing it is straightforward due to the clear relation with the *previous* edges. Section 4.4 describes this in more detail, also showing the algorithm used to find all dependencies.

Once the memory accessing dependencies are solved the separate accessing and execute units can be constructed. The accessing units depend on the number of memory accesses while there is only a single execute unit, that consist of the source code with the memory accessing replaced with streams coming from the accessing units. This leads into the research sub-question:

How can one establish correct communication between the different access and execute units?

The communication between the different unit is achieved by using queues. The specific type used by this thesis it the standard FIFO type. Section 4.5 describes the possibility of having the ability to have different types of queues.

To validate the resulting framework, an experimental setup is to be created that validates against other hardware implementations. The last research sub-question becomes:

How does the execution time compare against other hardware implementations (baseline benchmark, manually optimized)?

The evaluation of the framework shows that generally an improvement in speedup is observed while also an increase in area and power usage is observed. While for

some benchmarks the speedup was negligible (1.004x) for benchmarks where a higher degree of memory accessing decoupling is possible a higher speedup was observed (up to 1.63x). Chapter 6 describes in detail the exact speedup that is achievable, as it depends on the type of implementation the results vary.

To improve the speedup of an algorithm prior to using a HLS tool this thesis focused on the design and implementation of a framework that can automatically translate C/C++ code to the DAE architecture so that it can be optimized for an FPGA. This forms the main research question:

Which steps are required to automatically translate C/C++ code to efficient HLS code for FPGAs using the DAE paradigm?

The starting point for the framework, initial step, is to get the C/C++ code in an intermediate representation that is abstract enough for manipulation while not losing information. This intermediate representation encompasses a selected code block also adding structural information, so that an algorithm can traverse the source code. The memory accessing elements are identified based on how they are written in the C/C++ programming language. The framework specifically locates brackets with an index in between. The address of memory accessing elements is generated based on dependencies defined elsewhere in the code. These are solved for by reverse traversing the intermediate representation starting at the memory accessing element. Next the different DAE units are created. It consists of a single execute unit that contains the input source code in the IR form with its memory accessing elements replaced by streams. Then multiple accessing units are created which are only responsible for reading or writing data into memory. All these units run in parallel. Data is coming from or going to the execute unit via streams. Blocking FIFOs are used for this. A unit is automatically blocked when the data isn't available yet. This allows for the units to fetch data from memory while the execute unit is still performing computations. HLS tool specific directives are used to allow for parallelizing the units and pipelining the loops within the different units. Finally, the results are evaluated using well known benchmarks, an overall increase of speedup and chip area used is observed. More details and complete design flow are described in Chapter 4.

7.2 Recommendations

The framework implements the design as it was specified in Chapter 4. In that chapter limitations of the approach are identified in Section 4.8. For instance one assumption that it made is the use of read-only and write-only buffers, buffers are

never read-write capable. A future extension to this thesis would be the support for structures where a read after write to the same buffer location works as intended. Section 4.8 describes a solution would be to move all accessing of the same memory accessing element in a single unit, this could be implemented in the future, but another solution was also described where the data access read unit should first look into the address generation queue of the data access write unit for the same variable stalling the read queue until the data is written.

Another limitation is occurs when a memory accessing element is accessed multiple times. This results in the framework generating multiple accessing units for the same accessing element (at a different address). These access units run in parallel. This cannot be synthesized by the HLS tool as it would result in the memory access port connected to two units. A solution for future work could consist in moving the logic into a single access unit, this is the same solution as it was described in the previous paragraph. Another solution would be to increase the number of ports on the memory accessing element, this can be achieved by using two unique names for the memory accessing element. This results in two independent interfaces that can be connected to a memory element with two ports.

The moment a memory access depends on another memory access is not supported. The framework currently does not support connecting streams from access units to access units. Implementing this feature a future work fixes this limitation.

Section 5.1 introduced libclang and libTooling. LibTooling provides facilities for code refactoring, essentially applying manipulations on top of the parsed input code. Removing the burden of formatting when generating the output code, which reduces the overall code size of the framework. If the goal of human readable code is relaxed then the use of libTooling and its associated libraries might become a viable option for future work though this is at the cost of having code that may not be as well human readable as the code is parsed into the LLVM AST that removes some information from the code.

Recently, Xilinx has open-sourced the front end to its HLS tool (Vitis HLS) [27]. This adds the ability for the framework to be expanded to better integrate with it. Future work could consist of adding the architectural translation capabilities to the Xilinx HLS tool and also automatically adding the appropriate HLS directives. This approach doesn't produce new source code in a human readable form. Instead, it generates an intermediate representation specific for LLVM. An engineer could experiment on the IR generated for LLVM but because its much more abstract, remember it resembles a platform independent assembly code, experimenting could be much more time consuming. Another method would be to exclusively experiment with HLS directives directly on the source code that was provided as the input. As this is recent work, documentation on this is limited and further research is required.

The framework enables array partitioning for local arrays. It uses the *complete* method which turns the local array into local elements. This takes up the most area while attempting to increase the speedup. This approach may not always work for very large arrays, depending on the target FPGA the framework may be extended to support variable array partitioning so that an engineer can balance the area vs speedup depending on what is required for the engineer.

Bibliography

[1] A. Canis, J. Choi, B. Fort, B. Syrowik, R. L. Lian, Y. T. Chen, H. Hsiao, J. Goeders, S. Brown, and J. Anderson, *LegUp High-Level Synthesis*. Cham: Springer International Publishing, 2016, pp. 175–190. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_10

[2] G. Charitopoulos, C. Vatsolakis, G. Chrysos, and D. N. Pnevmatikatos, "A Decoupled Access-Execute Architecture for Reconfigurable Accelerators," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 244–247. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/3203217.3203267

[3] S. Kumar, "Fundamental Limits to Moore's Law," *arXiv e-prints*, p. arXiv:1511.05956, Nov. 2015. [Online]. Available: https://arxiv.org/abs/1511.05956

[4] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[5] "Introduction to FPGA Design with Vivado High-Level Synthesis (UG998)," https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf, accessed: 2021-07-05.

[6] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[7] K. Krommydas, W.-c. Feng, C. D. Antonopoulos, and N. Bellas, "OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures," *Journal of Signal Processing Systems*, vol. 85, no. 3, pp. 373–392, Dec 2016. [Online]. Available: https://doi.org/10.1007/s11265-015-1051-z

[8] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 110–119.

[9] "HLS Pragmas," https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/hls_pragmas.html, accessed: 2021-07-05.

[10] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *SIGARCH Comput. Archit. News*, vol. 10, no. 3, p. 112–119, Apr. 1982. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/1067649.801719

[11] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011. Citeseer, 2011, p. 1.

[12] G. Balogh, G. Mudalige, I. Reguly, S. Antao, and C. Bertolli, "OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2018, pp. 59–70.

[13] J. Cong, M. Huang, P. Pan, Y. Wang, and P. Zhang, *Source-to-Source Optimization for HLS*. Cham: Springer International Publishing, 2016, pp. 137–163. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_8

[14] N. Jacobsen, "LLVM supported source-to-source translation - Translation from annotated C/C++ to CUDA C/C++," Master's thesis, University of Oslo, 01 2016.

[15] D. Unat, "Domain-Specific Translator and Optimizer for Massive On-Chip Parallelism," phd, University of California, San Diego, 2012.

[16] N. Alachiotis, C. Vatsolakis, G. Chrysos, and D. Pnevmatikatos, "Raisd-x: A fast and accurate fpga system for the detection of positive selection in thousands of genomes," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 1, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3364225

[17] ——, "Accelerated inference of positive selection on whole genomes," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2018, pp. 202–2027.

[18] D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra, "CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/3358177

[19] LLVM, “The LLVM Compiler Infrastructure Project,” https://llvm.org/, accessed: 2021-10-02.

[20] ——, “Choosing the Right Interface for Your Application — Clang 13 documentation,” https://clang.llvm.org/docs/Tooling.html, accessed: 2021-10-02.

[21] Xilinx, “Vitis unified software platform,” https://www.xilinx.com/products/design-tools/vitis.html, accessed: 2021-08-10.

[22] ——, “Introduction to Vitis HLS,” https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/introductionvitishls.html, accessed: 2021-08-10.

[23] ——, “Xilinx Runtime Library (XRT),” https://www.xilinx.com/products/design-tools/vitis/xrt.html, accessed: 2021-08-10.

[24] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[25] Y. Saad, *3. Sparse Matrices*. Society for Industrial and Applied Mathematics, 2003, pp. 73–101. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.ch3

[26] Xilinx, “Burst Inference Failure 7,” https://www.xilinx.com/html_docs/xilinx2021_1/hls-guidance/214-229.html, accessed: 2021-10-02.

[27] ——, “Vitis HLS Front-end,” https://forums.xilinx.com/t5/AI-and-Machine-Learning-Blog/Opening-a-World-of-Possibilities-Vitis-HLS-Front-end-is-Now-Open/ba-p/1211207, accessed: 2021-08-10.

Appendix A

Example DAE translation

This chapter will show an example of an C/C++ code that is translated to the DAE architecture. The code sample shown in Section 4.3 will be used.

```
1  void fetch_unit0(int * m1, hls::stream<int> & m1_rs0) {
2    int i;
3    int i_row;
4    int j;
5    for(i = 0; i<col_size; i ++) {
6  #pragma HLS PIPELINE
7      i_row = i * row_size;
8      for(j = 0; j<row_size; j ++) {
9  #pragma HLS PIPELINE
10       m1_rs0.write(m1[i_row + j]);
11     }
12   }
13 }
14 void write_unit1(int * v1, hls::stream<int> & v1_ws1) {
15   int i;
16   for(i = 0; i<col_size; i ++) {
17 #pragma HLS PIPELINE
18     v1[i] = v1_ws1.read();
19   }
20 }
21 void process_unit(hls::stream<int> & m1_rs0, hls::stream<int> & v1_ws1) {
22   int i;
23   int sum;
24   int j;
25   for(i = 0; i<col_size; i ++) {
26 #pragma HLS PIPELINE
27     sum = 0;
28     for(j = 0; j<row_size; j ++) {
29 #pragma HLS PIPELINE
30       sum += m1_rs0.read();
31     }
```

```
32      v1_ws1.write( sum);
33    }
34  }
35  void vector_adder(int * m1, int * v1) {
36  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0 port=m1
37  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem1 port=v1
38    hls::stream<int> m1_rs0;
39  #pragma HLS STREAM variable=m1_rs0
40    hls::stream<int> v1_ws1;
41  #pragma HLS STREAM variable=v1_ws1
42  #pragma HLS DATAFLOW
43    fetch_unit0(m1, m1_rs0);
44    process_unit(m1_rs0, v1_ws1);
45    write_unit1(v1, v1_ws1);
46  }
```

Listing A.1: Matrix vector addition

Appendix B

MachSuite benchmarks

This chapter will show the code of the MachSuite benchmarks that are translated in the experimental chapter.

B.1 gemm

```
#include "gemm.h"

void gemm( TYPE *m1, TYPE *m2, TYPE *prod ){
#pragma HLS INTERFACE s_axilite port=m1
#pragma HLS INTERFACE s_axilite port=m2
#pragma HLS INTERFACE s_axilite port=prod
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0 port=m1
#pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem1 port=m2
#pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem2 port=prod
    int i, j, k;
    int k_col, i_col;
    TYPE mult;

    outer:for(i=0;i<row_size;i++) {
//#pragma HLS PIPELINE
        middle:for(j=0;j<col_size;j++) {
#pragma HLS PIPELINE
            i_col = i * col_size;
            TYPE sum = 0;
            inner:for(k=0;k<row_size;k++) {
#pragma HLS PIPELINE
                k_col = k * col_size;
                mult = m1[i_col + k] * m2[k_col + j];
                sum += mult;
            }
            prod[i_col + j]  = sum;
```

```
        }
    }
}
```

Listing B.1: gemm: Kernel original code

```
#include <stdlib.h>
#include <stdio.h>
#include <hls_stream.h>
#include "./gemm.h"
void fetch_unit0(double * m1, hls::stream<double> & m1_rs0) {
  int i;
  int j;
  int i_col;
  int k;
  int k_col;
  fetch_unit0_outer: for(i = 0; i<row_size; i ++) {
//#pragma HLS PIPELINE
    fetch_unit0_middle: for(j = 0; j<col_size; j ++) {
#pragma HLS PIPELINE
      i_col = i * col_size;
      fetch_unit0_inner: for(k = 0; k<row_size; k ++) {
#pragma HLS PIPELINE
        k_col = k * col_size;
        m1_rs0.write(m1[i_col + k]);
      }
    }
  }
}
void fetch_unit1(double * m2, hls::stream<double> & m2_rs1) {
  int i;
  int j;
  int i_col;
  int k;
  int k_col;
  fetch_unit1_outer: for(i = 0; i<row_size; i ++) {
//#pragma HLS PIPELINE
    fetch_unit1_middle: for(j = 0; j<col_size; j ++) {
#pragma HLS PIPELINE
      i_col = i * col_size;
      fetch_unit1_inner: for(k = 0; k<row_size; k ++) {
#pragma HLS PIPELINE
        k_col = k * col_size;
        m2_rs1.write(m2[k_col + j]);
      }
    }
  }
```

```
}
void write_unit2(double * prod, hls::stream<double> & prod_ws2) {
  int i;
  int j;
  int i_col;
  write_unit2_outer: for(i = 0; i<row_size; i ++) {
#pragma HLS PIPELINE
    write_unit2_middle: for(j = 0; j<col_size; j ++) {
#pragma HLS PIPELINE
      i_col = i * col_size;
      prod[i_col + j] = prod_ws2.read();
    }
  }
}
void process_unit(hls::stream<double> & m1_rs0, hls::stream<double> & m2_rs1,
↪  hls::stream<double> & prod_ws2) {
  int i;
  int j;
  int i_col;
  int k;
  int k_col;
  double mult;
  outer: for(i = 0; i<row_size; i ++) {
//#pragma HLS PIPELINE
    middle: for(j = 0; j<col_size; j ++) {
#pragma HLS PIPELINE
      i_col = i * col_size;
      TYPE sum = 0;
      inner: for(k = 0; k<row_size; k ++) {
#pragma HLS PIPELINE
        k_col = k * col_size;
        mult = m1_rs0.read() * m2_rs1.read();
        sum += mult;
      }
      prod_ws2.write( sum);
    }
  }
}
void gemm(double * m1, double * m2, double * prod) {
#pragma HLS INTERFACE s_axilite port=m1
#pragma HLS INTERFACE s_axilite port=m2
#pragma HLS INTERFACE s_axilite port=prod
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0 port=m1
#pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem1 port=m2
#pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem2 port=prod
  hls::stream<double> m1_rs0;
```

```
88 #pragma HLS STREAM variable=m1_rs0
89   hls::stream<double> m2_rs1;
90 #pragma HLS STREAM variable=m2_rs1
91   hls::stream<double> prod_ws2;
92 #pragma HLS STREAM variable=prod_ws2
93 #pragma HLS DATAFLOW
94   fetch_unit0(m1, m1_rs0);
95   fetch_unit1(m2, m2_rs1);
96   process_unit(m1_rs0, m2_rs1, prod_ws2);
97   write_unit2(prod, prod_ws2);
98 }
```

Listing B.2: gemm: Kernel translated code

```
1  #include "gemm.h"
2  #include <string.h>
3  #include "xcl2.hpp"
4  #include <vector>
5
6  #include <string.h>
7  #include <unistd.h>
8  #include <fcntl.h>
9  #include <sys/stat.h>
10 #include <assert.h>
11
12 #include <chrono>
13 using namespace std::chrono;
14
15 int INPUT_SIZE = sizeof(struct bench_args_t);
16
17 #define EPSILON ((TYPE)1.0e-6)
18
19
20 std::string binaryFile;
21
22 const int N = row_size*col_size;
23
24 void run_benchmark( void *vargs ) {
25   struct bench_args_t *args = (struct bench_args_t *)vargs;
26   //gemm( args->m1, args->m2, args->prod );
27
28   printf("Preparing accelerator\n");
29   cl_int err;
30   cl::Kernel krnl_add;
31   cl::CommandQueue q;
32   cl::Context context;
33
```

```
34    /*
35      TYPE m1[N];
36      TYPE m2[N];
37      TYPE prod[N];
38    */
39    std::vector<TYPE, aligned_allocator<TYPE> > m1(N);
40    std::vector<TYPE, aligned_allocator<TYPE> > m2(N);
41    std::vector<TYPE, aligned_allocator<TYPE> > prod(N);
42
43    for (int i=0; i <(N); i++) {
44      m1[i] = args->m1[i];
45      m2[i] = args->m2[i];
46      prod[i] = args->prod[i];
47    }
48
49
50
51    //gemm( args->m1, args->m2, args->prod );
52    // OPENCL HOST CODE AREA START
53    // get_xil_devices() is a utility API which will find the xilinx
54    // platforms and will return list of devices connected to Xilinx platform
55    auto devices = xcl::get_xil_devices();
56    // read_binary_file() is a utility API which will load the binaryFile
57    // and will return the pointer to file buffer.
58    auto fileBuf = xcl::read_binary_file(binaryFile);
59    cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
60    bool valid_device = false;
61    for (unsigned int i = 0; i < devices.size(); i++) {
62        auto device = devices[i];
63        // Creating Context and Command Queue for selected Device
64        OCL_CHECK(err, context = cl::Context(device, nullptr, nullptr, nullptr,
      ↪  &err));
65        OCL_CHECK(err, q = cl::CommandQueue(context, device,
      ↪  CL_QUEUE_PROFILING_ENABLE, &err));
66        std::cout << "Trying to program device[" << i << "]: " <<
      ↪  device.getInfo<CL_DEVICE_NAME>() << std::endl;
67        cl::Program program(context, {device}, bins, nullptr, &err);
68        if (err != CL_SUCCESS) {
69            std::cout << "Failed to program device[" << i << "] with xclbin file!\n";
70        } else {
71            std::cout << "Device[" << i << "]: program successful!\n";
72            OCL_CHECK(err, krnl_add = cl::Kernel(program, "gemm", &err));
73            valid_device = true;
74            break; // we break because we found a valid device
75        }
76    }
77    if (!valid_device) {
```

```
78        std::cout << "Failed to program any device found, exit!\n";
79        exit(EXIT_FAILURE);
80    }
81
82    // Allocate Buffer in Global Memory
83    // Buffers are allocated using CL_MEM_USE_HOST_PTR for efficient memory and
84    // Device-to-host communication
85
86    OCL_CHECK(err, cl::Buffer buffer_i1(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(TYPE) * (N),
87                                        m1.data(), &err));
88    OCL_CHECK(err, cl::Buffer buffer_i2(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(TYPE) * (N),
89                                        m2.data(), &err));
90
91    OCL_CHECK(err, cl::Buffer buffer_o3(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_WRITE_ONLY, sizeof(TYPE) * (N),
92                                        prod.data(), &err));
93
94    // Default version
95    OCL_CHECK(err, err = krnl_add.setArg(0, buffer_i1));
96    OCL_CHECK(err, err = krnl_add.setArg(1, buffer_i2));
97    OCL_CHECK(err, err = krnl_add.setArg(2, buffer_o3));
98
99    // Copy input data to device global memory
100   OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_i1, buffer_i2}, 0 /* 0
      ↪  means from host*/));
101   OCL_CHECK(err, err = q.finish());
102
103   // Launch the Kernel
104   // For HLS kernels global and local size is always (1,1,1). So, it is
105   // recommended
106   // to always use enqueueTask() for invoking HLS kernel
107   std::cout << "Starting kernel\n";
108   auto start = high_resolution_clock::now();
109   OCL_CHECK(err, err = q.enqueueTask(krnl_add));
110   std::cout << "Waiting for accelerator to finish\n";
111   OCL_CHECK(err, err = q.finish());
112   std::cout << "Waiting for accelerator to finish2\n";
113   // Copy Result from Device Global Memory to Host Local Memory
114   //OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_aligna, buffer_alignb,
      ↪  buffer_m, buffer_ptr}, CL_MIGRATE_MEM_OBJECT_HOST));
115   OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_o3},
      ↪  CL_MIGRATE_MEM_OBJECT_HOST));
116   std::cout << "Waiting for accelerator to finish3\n";
117   OCL_CHECK(err, err = q.finish());
118   std::cout << "Waiting for accelerator to finish4\n";
```

```
119   // OPENCL HOST CODE AREA END
120
121   auto stop = high_resolution_clock::now();
122   auto duration = duration_cast<microseconds>(stop - start);
123
124   std::cout << "Execution time: " << duration.count() << std::endl;
125
126   printf("Fetching data from accelerator\n");
127
128
129   //printf("Data fetched01\n");
130
131   for (int i=0; i <(N); i++) {
132     args->m1[i] = m1[i];
133     args->m2[i] = m2[i];
134     args->prod[i] = prod[i];
135   }
136
137   //printf("%f ?= %f\r\n", args->m1[0], args->prod[0]);
138   printf("Data fetched10\n");
139
140
141
142
143   // Parse command line.
144   const char *check_file = "data/check.data";
145
146   char *data = (char*)vargs;
147
148   // Load check data
149   printf("Checking output\n");
150   int check_fd;
151   char *ref;
152   ref = (char*) malloc(INPUT_SIZE);
153   assert( ref!=NULL && "Out of memory" );
154   check_fd = open( check_file, O_RDONLY );
155   assert( check_fd>0 && "Couldn't open check data file");
156   output_to_data(check_fd, ref);
157
158   // Validate benchmark results
159   printf("Validating output\n");
160   if( !check_data(data, ref) ) {
161     fprintf(stderr, "Benchmark results are incorrect\n");
162     //return -1;
163   } else {
164     fprintf(stderr, "BENCH SUCCESS!\n");
165   }
```

```
166   printf("Free!\n");
167   //free(data);
168   free(ref);
169
170   printf("Success.\n");
171   exit(0);
172
173 }
174
175 /* Input format:
176 %% Section 1
177 TYPE[N]: matrix 1
178 %% Section 2
179 TYPE[N]: matrix 2
180 */
181
182 void input_to_data(int fd, void *vdata) {
183   struct bench_args_t *data = (struct bench_args_t *)vdata;
184   char *p, *s;
185   // Zero-out everything.
186   memset(vdata,0,sizeof(struct bench_args_t));
187   // Load input string
188   p = readfile(fd);
189
190   s = find_section_start(p,1);
191   STAC(parse_,TYPE,_array)(s, data->m1, N);
192
193   s = find_section_start(p,2);
194   STAC(parse_,TYPE,_array)(s, data->m2, N);
195   free(p);
196
197 }
198
199 void data_to_input(int fd, void *vdata) {
200   struct bench_args_t *data = (struct bench_args_t *)vdata;
201
202   write_section_header(fd);
203   STAC(write_,TYPE,_array)(fd, data->m1, N);
204
205   write_section_header(fd);
206   STAC(write_,TYPE,_array)(fd, data->m2, N);
207 }
208
209 /* Output format:
210 %% Section 1
211 TYPE[N]: output matrix
212 */
```

```

void output_to_data(int fd, void *vdata) {
  struct bench_args_t *data = (struct bench_args_t *)vdata;
  char *p, *s;
  // Load input string
  p = readfile(fd);

  s = find_section_start(p,1);
  STAC(parse_,TYPE,_array)(s, data->prod, N);
  free(p);
}

void data_to_output(int fd, void *vdata) {
  struct bench_args_t *data = (struct bench_args_t *)vdata;

  write_section_header(fd);
  STAC(write_,TYPE,_array)(fd, data->prod, N);
}

int check_data( void *vdata, void *vref ) {
  struct bench_args_t *data = (struct bench_args_t *)vdata;
  struct bench_args_t *ref = (struct bench_args_t *)vref;
  int has_errors = 0;
  int r,c;
  TYPE diff;

  for( r=0; r<row_size; r++ ) {
    for( c=0; c<col_size; c++ ) {
      diff = data->prod[r*row_size + c] - ref->prod[r*row_size+c];
      has_errors |= (diff<-EPSILON) || (EPSILON<diff);
    }
  }

  // Return true if it's correct.
  return !has_errors;
}
```

Listing B.3: gemm: Host code

B.2 spmv

```
/*
Based on algorithm described here:
http://www.cs.berkeley.edu/~mhoemmen/matrix-seminar/slides/UCB_sparse_tutorial_1.pdf
*/
```

```
5
6  #include "spmv.h"
7  #include <hls_stream.h>
8
9  void spmv(double * val, int32_t *cols, int32_t * rowDelimiters,  double * vec,
   ↪  double * out){
10 #pragma HLS INTERFACE s_axilite port=val
11 #pragma HLS INTERFACE s_axilite port=cols
12 #pragma HLS INTERFACE s_axilite port=rowDelimiters
13 #pragma HLS INTERFACE s_axilite port=vec
14 #pragma HLS INTERFACE s_axilite port=out
15 #pragma HLS INTERFACE s_axilite port=return
16 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 port=val bundle=gmem0
17 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 port=cols bundle=gmem1
18 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 port=rowDelimiters
   ↪  bundle=gmem2
19 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 port=vec bundle=gmem3
20 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 port=out bundle=gmem4
21     int i, j;
22     TYPE sum, Si;
23
24     spmv_1 : for(i = 0; i < N; i++){
25 #pragma HLS PIPELINE
26         sum = 0; Si = 0;
27         int tmp_begin = rowDelimiters[i];
28         int tmp_end = rowDelimiters[i+1];
29         spmv_2 : for (j = tmp_begin; j < tmp_end; j++){
30 #pragma HLS PIPELINE
31             Si = val[j] * vec[cols[j]];
32             sum = sum + Si;
33         }
34         out[i] = sum;
35     }
36 }
```

Listing B.4: spmv: Kernel original code

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <hls_stream.h>
4  #include "./spmv.h"
5
6  void fetch_unit1(int32_t * rowDelimiters, hls::stream<int32_t> & rowDelimiters_rs0,
   ↪  hls::stream<int32_t> & rowDelimiters1_rs1) {
7    int i;
8    fetch_unit1_spmv_1: for(i = 0; i<N; i ++) {
9 #pragma HLS PIPELINE
```

```
10      rowDelimiters_rs0.write(rowDelimiters[i]);
11      rowDelimiters1_rs1.write(rowDelimiters[i + 1]);
12    }
13  }
14  void fetch_unit2(double * val, hls::stream<double> & val_rs2,
15                   hls::stream<int32_t> & rowDelimiters_rs0, hls::stream<int32_t> &
    ↪  rowDelimiters1_rs1,
16                   hls::stream<int32_t> & rowDelimiters_rs2, hls::stream<int32_t> &
    ↪  rowDelimiters1_rs3) {
17    int i;
18    int j;
19    fetch_unit2_spmv_1: for(i = 0; i<N; i ++) {
20  #pragma HLS PIPELINE
21      int tmp_begin = rowDelimiters_rs0.read();
22      int tmp_end = rowDelimiters1_rs1.read();
23      rowDelimiters_rs2 << tmp_begin;
24      rowDelimiters1_rs3 << tmp_end;
25      fetch_unit2_spmv_2: for(j = tmp_begin; j<tmp_end; j ++) {
26  #pragma HLS PIPELINE
27        val_rs2.write(val[j]);
28      }
29    }
30  }
31  void fetch_unit3_1(int32_t* cols, hls::stream<int32_t> & cols_rs3,
32                   hls::stream<int32_t> & rowDelimiters_rs0, hls::stream<int32_t> &
    ↪  rowDelimiters1_rs1,
33                   hls::stream<int32_t> & rowDelimiters_rs2, hls::stream<int32_t> &
    ↪  rowDelimiters1_rs3) {
34    int i;
35    int j;
36    fetch_unit3_spmv_1: for(i = 0; i<N; i ++) {
37  #pragma HLS PIPELINE
38      int tmp_begin = rowDelimiters_rs0.read();
39      int tmp_end = rowDelimiters1_rs1.read();
40      rowDelimiters_rs2 << tmp_begin;
41      rowDelimiters1_rs3 << tmp_end;
42      fetch_unit3_spmv_2: for(j = tmp_begin; j<tmp_end; j ++) {
43  #pragma HLS PIPELINE
44        cols_rs3.write(cols[j]);
45      }
46    }
47  }
48  void fetch_unit3(double * vec, hls::stream<int32_t> & cols_rs3, hls::stream<double>
    ↪  & vec_rs3,
49                   hls::stream<int32_t> & rowDelimiters_rs0, hls::stream<int32_t> &
    ↪  rowDelimiters1_rs1,
```

```
50                  hls::stream<int32_t> & rowDelimiters_rs2, hls::stream<int32_t> &
    ↪  rowDelimiters1_rs3) {
51    int i;
52    int j;
53    fetch_unit3_spmv_1: for(i = 0; i<N; i ++) {
54  #pragma HLS PIPELINE
55      int tmp_begin = rowDelimiters_rs0.read();
56      int tmp_end = rowDelimiters1_rs1.read();
57      rowDelimiters_rs2 << tmp_begin;
58      rowDelimiters1_rs3 << tmp_end;
59      fetch_unit3_spmv_2: for(j = tmp_begin; j<tmp_end; j ++) {
60  #pragma HLS PIPELINE
61        vec_rs3.write(vec[cols_rs3.read()]);
62      }
63    }
64  }
65  void write_unit4(double * out, hls::stream<double> & out_ws4) {
66    int i;
67    int j;
68    write_unit4_spmv_1: for(i = 0; i<N; i ++) {
69  #pragma HLS PIPELINE
70      out[i] = out_ws4.read();
71    }
72  }
73  void process_unit(hls::stream<int32_t> & rowDelimiters_rs0, hls::stream<int32_t> &
    ↪  rowDelimiters1_rs1, hls::stream<double> & val_rs2, hls::stream<double> &
    ↪  vec_rs3, hls::stream<double> & out_ws4) {
74    int i;
75    double sum;
76    int j;
77    spmv_1: for(i = 0; i<N; i ++) {
78  #pragma HLS PIPELINE
79      sum = 0;
80      int tmp_begin = rowDelimiters_rs0.read();
81      int tmp_end = rowDelimiters1_rs1.read();
82      spmv_2: for(j = tmp_begin; j<tmp_end; j ++) {
83  #pragma HLS PIPELINE
84        sum += val_rs2.read() * vec_rs3.read();
85      }
86      out_ws4.write( sum);
87    }
88  }
89  void spmv1(int32_t * rowDelimiters, double * val, double * vec, double * out,
    ↪  int32_t *cols) {
90  #pragma HLS INTERFACE s_axilite port=rowDelimiters
91  #pragma HLS INTERFACE s_axilite port=val
92  #pragma HLS INTERFACE s_axilite port=vec
```

```
93  #pragma HLS INTERFACE s_axilite port=out
94  #pragma HLS INTERFACE s_axilite port=cols
95  #pragma HLS INTERFACE s_axilite port=return
96  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0
    ↪  port=rowDelimiters
97  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem1 port=val
98  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem2 port=vec
99  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem3 port=out
100 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem4 port=cols
101   hls::stream<int32_t> rowDelimiters_rs0;
102 #pragma HLS STREAM variable=rowDelimiters_rs0
103   hls::stream<int32_t> rowDelimiters1_rs1;
104 #pragma HLS STREAM variable=rowDelimiters1_rs1
105
106   hls::stream<int32_t> rowDelimiters_rs2;
107 #pragma HLS STREAM variable=rowDelimiters_rs2
108   hls::stream<int32_t> rowDelimiters1_rs3;
109 #pragma HLS STREAM variable=rowDelimiters1_rs3
110
111   hls::stream<int32_t> rowDelimiters_rs4;
112 #pragma HLS STREAM variable=rowDelimiters_rs4
113   hls::stream<int32_t> rowDelimiters1_rs5;
114 #pragma HLS STREAM variable=rowDelimiters1_rs5
115
116   hls::stream<int32_t> rowDelimiters_rs6;
117 #pragma HLS STREAM variable=rowDelimiters_rs6
118   hls::stream<int32_t> rowDelimiters1_rs7;
119 #pragma HLS STREAM variable=rowDelimiters1_rs7
120
121   hls::stream<double> val_rs2;
122 #pragma HLS STREAM variable=val_rs2
123   hls::stream<double> vec_rs3;
124 #pragma HLS STREAM variable=vec_rs3
125   hls::stream<int32_t> cols_rs3;
126 #pragma HLS STREAM variable=cols_rs3
127   hls::stream<double> out_ws4;
128 #pragma HLS STREAM variable=out_ws4
129 #pragma HLS DATAFLOW
130   fetch_unit1(rowDelimiters, rowDelimiters_rs0, rowDelimiters1_rs1);
131   fetch_unit2(val, val_rs2, rowDelimiters_rs0, rowDelimiters1_rs1,
    ↪  rowDelimiters_rs2, rowDelimiters1_rs3);
132   fetch_unit3_1(cols, cols_rs3, rowDelimiters_rs2, rowDelimiters1_rs3,
    ↪  rowDelimiters_rs4, rowDelimiters1_rs5);
133   fetch_unit3(vec, cols_rs3, vec_rs3, rowDelimiters_rs4, rowDelimiters1_rs5,
    ↪  rowDelimiters_rs6, rowDelimiters1_rs7);
134   process_unit(rowDelimiters_rs6, rowDelimiters1_rs7, val_rs2, vec_rs3, out_ws4);
135   write_unit4(out, out_ws4);
```

```
136 }
```

Listing B.5: spmv: Kernel translated code

```
1  /*
2  Based on algorithm described here:
3  http://www.cs.berkeley.edu/~mhoemmen/matrix-seminar/slides/UCB_sparse_tutorial_1.pdf
4  */
5
6  #include "spmv.h"
7  #include <hls_stream.h>
8
9  void fetch_1(int32_t *rowDelimiters, hls::stream<int32_t> & rowDelimiters_rs0,
   ↪  hls::stream<int32_t> & rowDelimiters1_rs1) {
10     int i;
11     hls::stream<int> row_ptr_address;
12     hls::stream<int> row_ptr_next_address;
13 #pragma HLS DATAFLOW
14     fetch_1_1: for(i = 0; i < N; i++){
15 #pragma HLS PIPELINE
16         row_ptr_address << i;
17     }
18     fetch_1_2: for(i = 0; i < N; i++){
19 #pragma HLS PIPELINE
20         row_ptr_next_address << (i+1);
21     }
22     fetch_1_3: for(i = 0; i < N; i++){
23 #pragma HLS PIPELINE
24         rowDelimiters_rs0.write(rowDelimiters[row_ptr_address.read()]);
25         rowDelimiters1_rs1.write(rowDelimiters[row_ptr_next_address.read()]);
26     }
27 }
28
29 void fetch_2(int32_t *cols, hls::stream<int32_t> & row_ptr_data,
   ↪  hls::stream<int32_t> & row_ptr_next_data, hls::stream<int32_t> &
   ↪  row_ptr_data_out, hls::stream<int32_t> & row_ptr_next_data_out,
   ↪  hls::stream<int32_t> & col_ind_data) {
30     int i, j;
31     hls::stream<int> col_ind_address;
32     for(i = 0; i < N; i++){
33 #pragma HLS DATAFLOW
34         int beg = row_ptr_data.read();
35         int end = row_ptr_next_data.read();
36         row_ptr_data_out << beg;
37         row_ptr_next_data_out << end;
38         for (j = beg; j < end; j++){
39 #pragma HLS PIPELINE
```

```
40              col_ind_address << j;
41          }
42          for (j = beg; j < end; j++){
43  #pragma HLS PIPELINE
44              col_ind_data << cols[col_ind_address.read()];
45          }
46      }
47  }
48
49  void fetch_3(TYPE *val, TYPE *vec, hls::stream<int32_t> & row_ptr_data,
    ↪  hls::stream<int32_t> & row_ptr_next_data,
50               hls::stream<int32_t> & row_ptr_data_out, hls::stream<int32_t> &
    ↪  row_ptr_next_data_out, hls::stream<int32_t> & col_ind_data,
51               hls::stream<double> & vec_data, hls::stream<double> & val_data) {
52      int i, j;
53      hls::stream<int> val_address;
54      hls::stream<int> vec_address;
55      for(i = 0; i < N; i++){
56  #pragma HLS DATAFLOW
57          int beg = row_ptr_data.read();
58          int end = row_ptr_next_data.read();
59          row_ptr_data_out << beg;
60          row_ptr_next_data_out << end;
61          for (j = beg; j < end; j++){
62  #pragma HLS PIPELINE
63              val_address << j;
64              vec_address << col_ind_data.read();
65          }
66          for (j = beg; j < end; j++){
67  #pragma HLS PIPELINE
68              val_data << val[val_address.read()];
69              vec_data << vec[vec_address.read()];
70          }
71      }
72  }
73
74  void process(hls::stream<int32_t> & row_ptr_data, hls::stream<int32_t> &
    ↪  row_ptr_next_data, hls::stream<double> & vec_data, hls::stream<double> &
    ↪  val_data, hls::stream<double> & out_data) {
75      int i, j;
76      TYPE sum;
77      for(i = 0; i < N; i++){
78  #pragma HLS PIPELINE
79          int beg = row_ptr_data.read();
80          int end = row_ptr_next_data.read();
81          sum = 0;
82          for (j = beg; j < end; j++){
```

```
83  #pragma HLS PIPELINE
84              sum += vec_data.read() * val_data.read();
85          }
86          out_data << sum;
87      }
88  }
89
90  void fetch_4(TYPE* out, hls::stream<double> & out_data) {
91      hls::stream<int> out_address;
92  #pragma HLS DATAFLOW
93      for(int i = 0; i < N; i++){
94  #pragma HLS PIPELINE
95          out_address << i;
96      }
97      for(int i = 0; i < N; i++){
98  #pragma HLS PIPELINE
99          out[out_address.read()] = out_data.read();
100     }
101 }
102
103 void spmv(double * val, int32_t *cols, int32_t * rowDelimiters,  double * vec,
    ↪  double * out) {
104 #pragma HLS INTERFACE s_axilite port=val
105 #pragma HLS INTERFACE s_axilite port=cols
106 #pragma HLS INTERFACE s_axilite port=rowDelimiters
107 #pragma HLS INTERFACE s_axilite port=vec
108 #pragma HLS INTERFACE s_axilite port=out
109 #pragma HLS INTERFACE s_axilite port=return
110 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0 port=val
111 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem1 port=cols
112 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem2
    ↪  port=rowDelimiters
113 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem3 port=vec
114 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem4 port=out
115     hls::stream<int32_t> row_ptr_data;
116     hls::stream<int32_t> row_ptr_next_data;
117     hls::stream<int32_t> row_ptr_data1;
118     hls::stream<int32_t> row_ptr_next_data1;
119     hls::stream<int32_t> row_ptr_data2;
120     hls::stream<int32_t> row_ptr_next_data2;
121     hls::stream<int32_t> col_ind_data;
122     hls::stream<double> val_data;
123     hls::stream<double> vec_data;
124     hls::stream<double> out_data;
125 #pragma HLS DATAFLOW
126     fetch_1(rowDelimiters, row_ptr_data, row_ptr_next_data);
```

```
127     fetch_2(cols, row_ptr_data, row_ptr_next_data, row_ptr_data1,
    ↪   row_ptr_next_data1, col_ind_data);
128     fetch_3(val, vec, row_ptr_data1, row_ptr_next_data1, row_ptr_data2,
    ↪   row_ptr_next_data2, col_ind_data, vec_data, val_data);
129     process(row_ptr_data2, row_ptr_next_data2, vec_data, val_data, out_data);
130     fetch_4(out, out_data);
131 }
```

Listing B.6: spmv: Kernel translated optimized code

```
1  #include "spmv.h"
2  #include <string.h>
3  #include "xcl2.hpp"
4  #include <vector>
5
6  #include <string.h>
7  #include <unistd.h>
8  #include <fcntl.h>
9  #include <sys/stat.h>
10 #include <assert.h>
11
12 #include <chrono>
13 using namespace std::chrono;
14
15 int INPUT_SIZE = sizeof(struct bench_args_t);
16
17 std::string binaryFile;
18
19 #define EPSILON ((TYPE)1.0e-6)
20
21 void run_benchmark_daer( void *vargs, bool ex );
22
23 void run_benchmark_gen( void *vargs, bool ex) {
24   struct bench_args_t *args = (struct bench_args_t *)vargs;
25
26   printf("Preparing accelerator\r\n");
27
28   cl_int err;
29   cl::Kernel krnl_add;
30   cl::CommandQueue q;
31   cl::Context context;
32
33   std::vector<TYPE, aligned_allocator<TYPE> > source_val(NNZ);
34   std::vector<int32_t, aligned_allocator<int32_t> > source_cols(NNZ);
35   std::vector<int32_t, aligned_allocator<int32_t> > source_rowDelimiters(N+1);
36   //std::vector<int32_t, aligned_allocator<int32_t> > source_rowDelimiters1(N+1);
37   std::vector<TYPE, aligned_allocator<TYPE> > source_vec(N);
```

```
38    std::vector<TYPE, aligned_allocator<TYPE> > target_out(N);
39
40
41    for (int i=0; i < NNZ; i++) {
42      source_val[i] = args->val[i];
43      source_cols[i] = args->cols[i];
44    }
45    for (int i=0; i < N+1; i++) {
46      source_rowDelimiters[i] = args->rowDelimiters[i];
47      //source_rowDelimiters1[i] = args->rowDelimiters[i];
48    }
49    for (int i=0; i < N; i++) {
50      source_vec[i] = args->vec[i];
51      target_out[i] = args->out[i];
52    }
53
54    auto start = high_resolution_clock::now();
55    //gemm( args->m1, args->m2, args->prod );
56    // OPENCL HOST CODE AREA START
57    // get_xil_devices() is a utility API which will find the xilinx
58    // platforms and will return list of devices connected to Xilinx platform
59    auto devices = xcl::get_xil_devices();
60    // read_binary_file() is a utility API which will load the binaryFile
61    // and will return the pointer to file buffer.
62    auto fileBuf = xcl::read_binary_file(binaryFile);
63    cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
64    bool valid_device = false;
65    for (unsigned int i = 0; i < devices.size(); i++) {
66        auto device = devices[i];
67        // Creating Context and Command Queue for selected Device
68        OCL_CHECK(err, context = cl::Context(device, nullptr, nullptr, nullptr,
      ↪  &err));
69        OCL_CHECK(err, q = cl::CommandQueue(context, device,
      ↪  CL_QUEUE_PROFILING_ENABLE, &err));
70        std::cout << "Trying to program device[" << i << "]: " <<
      ↪  device.getInfo<CL_DEVICE_NAME>() << std::endl;
71        cl::Program program(context, {device}, bins, nullptr, &err);
72        if (err != CL_SUCCESS) {
73            std::cout << "Failed to program device[" << i << "] with xclbin file!\n";
74        } else {
75            std::cout << "Device[" << i << "]: program successful!\n";
76            OCL_CHECK(err, krnl_add = cl::Kernel(program, "spmv1", &err));
77            valid_device = true;
78            break; // we break because we found a valid device
79        }
80    }
81    if (!valid_device) {
```

```
82        std::cout << "Failed to program any device found, exit!\n";
83        exit(EXIT_FAILURE);
84    }
85
86    // Allocate Buffer in Global Memory
87    // Buffers are allocated using CL_MEM_USE_HOST_PTR for efficient memory and
88    // Device-to-host communication
89
90    OCL_CHECK(err, cl::Buffer buffer_v1(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(TYPE) * NNZ,
91                                        source_val.data(), &err));
92    OCL_CHECK(err, cl::Buffer buffer_v2(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(int32_t) * NNZ,
93                                        source_cols.data(), &err));
94    OCL_CHECK(err, cl::Buffer buffer_v3(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(int32_t) * (N+1),
95                                        source_rowDelimiters.data(), &err));
96    /*OCL_CHECK(err, cl::Buffer buffer_v6(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(int32_t) * (N+1),
97                                        source_rowDelimiters1.data(), &err));*/
98    OCL_CHECK(err, cl::Buffer buffer_v4(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(TYPE) * N,
99                                        source_vec.data(), &err));
100
101   OCL_CHECK(err, cl::Buffer buffer_v5(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_WRITE_ONLY, sizeof(TYPE) * N,
102                                       target_out.data(), &err));
103   /*OCL_CHECK(err, cl::Buffer buffer_m(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_WRITE, sizeof(int) * (ALEN+1)*(BLEN+1),
104                                       rw_m.data(), &err));
105   OCL_CHECK(err, cl::Buffer buffer_ptr(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_WRITE, sizeof(char) * (ALEN+1)*(BLEN+1),
106                                       rw_ptr.data(), &err));*/
107
108   //int size = DATA_SIZE;
109   //int32_t * rowDelimiters, int32_t * rowDelimiters1, double * val, double * vec,
      ↪  double * out, cols
110   /* Generated version */
111   OCL_CHECK(err, err = krnl_add.setArg(0, buffer_v3));
112   //OCL_CHECK(err, err = krnl_add.setArg(1, buffer_v6));
113   OCL_CHECK(err, err = krnl_add.setArg(1, buffer_v1));
114   OCL_CHECK(err, err = krnl_add.setArg(2, buffer_v4));
115   OCL_CHECK(err, err = krnl_add.setArg(3, buffer_v5));
116   OCL_CHECK(err, err = krnl_add.setArg(4, buffer_v2));
117   /**/
118
119   /* Default version
```

```
120   OCL_CHECK(err, err = krnl_add.setArg(0, buffer_v1));
121   OCL_CHECK(err, err = krnl_add.setArg(1, buffer_v2));
122   OCL_CHECK(err, err = krnl_add.setArg(2, buffer_v3));
123   OCL_CHECK(err, err = krnl_add.setArg(3, buffer_v4));
124   OCL_CHECK(err, err = krnl_add.setArg(4, buffer_v5));
125   /**/
126
127   //OCL_CHECK(err, err = krnl_add.setArg(1, buffer_seqb));
128   //OCL_CHECK(err, err = krnl_add.setArg(2, buffer_aligna));
129   //OCL_CHECK(err, err = krnl_add.setArg(3, buffer_alignb));
130   /*OCL_CHECK(err, err = krnl_add.setArg(4, buffer_m));
131   OCL_CHECK(err, err = krnl_add.setArg(5, buffer_m));
132   OCL_CHECK(err, err = krnl_add.setArg(6, buffer_m));
133   OCL_CHECK(err, err = krnl_add.setArg(7, buffer_m));
134   OCL_CHECK(err, err = krnl_add.setArg(8, buffer_ptr));*/
135
136   // Copy input data to device global memory
137   OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_v1, buffer_v2, buffer_v3,
      ↪  buffer_v4/*, buffer_v6*/}, 0 /* 0 means from host*/));
138   OCL_CHECK(err, err = q.finish());
139
140   // Launch the Kernel
141   // For HLS kernels global and local size is always (1,1,1). So, it is
142   // recommended
143   // to always use enqueueTask() for invoking HLS kernel
144   std::cout << "Starting kernel\n";
145   OCL_CHECK(err, err = q.enqueueTask(krnl_add));
146   std::cout << "Waiting for accelerator to finish\n";
147   OCL_CHECK(err, err = q.finish());
148   std::cout << "Waiting for accelerator to finish2\n";
149   // Copy Result from Device Global Memory to Host Local Memory
150   //OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_aligna, buffer_alignb,
      ↪  buffer_m, buffer_ptr}, CL_MIGRATE_MEM_OBJECT_HOST));
151   OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_v5},
      ↪  CL_MIGRATE_MEM_OBJECT_HOST));
152   std::cout << "Waiting for accelerator to finish3\n";
153   OCL_CHECK(err, err = q.finish());
154   std::cout << "Waiting for accelerator to finish4\n";
155   // OPENCL HOST CODE AREA END
156
157   auto stop = high_resolution_clock::now();
158   auto duration = duration_cast<microseconds>(stop - start);
159
160   std::cout << "Execution time: " << duration.count() << std::endl;
161
162   printf("Fetching data from accelerator %f\n", target_out[0]);
163
```

```
164
165   //printf("Data fetched01\n");
166
167
168   for (int i=0; i < NNZ; i++) {
169     args->val[i] = source_val[i];
170     args->cols[i] = source_cols[i];
171   }
172   for (int i=0; i < (N+1); i++) {
173     args->rowDelimiters[i] = source_rowDelimiters[i];
174   }
175   for (int i=0; i < N; i++) {
176     args->vec[i] = source_vec[i];
177     args->out[i] = target_out[i];
178   }
179
180   /*for (int i=0; i < ALEN; i++) {
181     args->seqA[i] = source_seqa[i];
182   }
183   printf("Data fetched02\n");
184   for (int i=0; i < BLEN; i++) {
185     args->seqB[i] = source_seqb[i];
186   }
187   printf("Data fetched03\n");
188   for (int i=0; i < (ALEN+BLEN); i++) {
189     args->alignedA[i] = target_aligna[i];
190     args->alignedB[i] = target_alignb[i];
191   }*/
192   /*printf("Data fetched04\n");
193   for (int i=0; i < ((ALEN+1)*(BLEN+1)); i++) {
194     args->M[i] = rw_m[i];
195     args->ptr[i] = rw_ptr[i];
196   }*/
197
198   //printf("%f ?= %f\r\n", args->m1[0], args->prod[0]);
199   printf("Data fetched10\n");
200
201
202
203
204   // Parse command line.
205   const char *check_file = "data/check.data";
206
207   char *data = (char*)vargs;
208
209   // Load check data
210   printf("Checking output\n");
```

```
211   int check_fd;
212   char *ref;
213   ref = (char*) malloc(INPUT_SIZE);
214   assert( ref!=NULL && "Out of memory" );
215   check_fd = open( check_file, O_RDONLY );
216   assert( check_fd>0 && "Couldn't open check data file");
217   output_to_data(check_fd, ref);
218
219   // Validate benchmark results
220   printf("Validating output\n");
221   if( !check_data(data, ref) ) {
222     fprintf(stderr, "Benchmark results are incorrect\n");
223     //return -1;
224   } else {
225     fprintf(stderr, "BENCH SUCCESS!\n");
226   }
227   printf("Free!\n");
228   //free(data);
229   free(ref);
230
231   printf("Success.\n");
232
233   if (ex) {
234     run_benchmark_daer(vargs, false);
235   }
236   exit(0);
237 }
238
239
240
241 void run_benchmark_daer( void *vargs, bool ex ) {
242   struct bench_args_t *args = (struct bench_args_t *)vargs;
243
244   printf("Preparing accelerator\r\n");
245
246 //  std::string binaryFile = "gemm.xclbin";
247
248   //int size = row_size*col_size;
249   cl_int err;
250   cl::Kernel krnl_add;
251   cl::CommandQueue q;
252   cl::Context context;
253   // Allocate Memory in Host Memory
254   //size_t vector_size_bytes = sizeof(TYPE) * size;
255   //printf("Buffer sizes: %d\r\n", size);
256
257   /*
```

```
258
259   TYPE val[NNZ];
260   int32_t cols[NNZ];
261   int32_t rowDelimiters[N+1];
262   TYPE vec[N];
263   TYPE out[N];
264   */
265
266
267   std::vector<TYPE, aligned_allocator<TYPE> > source_val(NNZ);
268   std::vector<int32_t, aligned_allocator<int32_t> > source_cols(NNZ);
269   std::vector<int32_t, aligned_allocator<int32_t> > source_rowDelimiters(N+1);
270   std::vector<TYPE, aligned_allocator<TYPE> > source_vec(N);
271   std::vector<TYPE, aligned_allocator<TYPE> > target_out(N);
272
273
274   for (int i=0; i < NNZ; i++) {
275     source_val[i] = args->val[i];
276     source_cols[i] = args->cols[i];
277   }
278   for (int i=0; i < N+1; i++) {
279     source_rowDelimiters[i] = args->rowDelimiters[i];
280   }
281   for (int i=0; i < N; i++) {
282     source_vec[i] = args->vec[i];
283     target_out[i] = args->out[i];
284   }
285
286   auto start = high_resolution_clock::now();
287   //gemm( args->m1, args->m2, args->prod );
288   // OPENCL HOST CODE AREA START
289   // get_xil_devices() is a utility API which will find the xilinx
290   // platforms and will return list of devices connected to Xilinx platform
291   auto devices = xcl::get_xil_devices();
292   // read_binary_file() is a utility API which will load the binaryFile
293   // and will return the pointer to file buffer.
294   auto fileBuf = xcl::read_binary_file(binaryFile);
295   cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
296   bool valid_device = false;
297   for (unsigned int i = 0; i < devices.size(); i++) {
298       auto device = devices[i];
299       // Creating Context and Command Queue for selected Device
300       OCL_CHECK(err, context = cl::Context(device, nullptr, nullptr, nullptr,
    ↪  &err));
301       OCL_CHECK(err, q = cl::CommandQueue(context, device,
    ↪  CL_QUEUE_PROFILING_ENABLE, &err));
```

```
302       std::cout << "Trying to program device[" << i << "]: " <<
    ↪  device.getInfo<CL_DEVICE_NAME>() << std::endl;
303       cl::Program program(context, {device}, bins, nullptr, &err);
304       if (err != CL_SUCCESS) {
305           std::cout << "Failed to program device[" << i << "] with xclbin file!\n";
306       } else {
307           std::cout << "Device[" << i << "]: program successful!\n";
308           OCL_CHECK(err, krnl_add = cl::Kernel(program, "spmv", &err));
309           valid_device = true;
310           break; // we break because we found a valid device
311       }
312   }
313   if (!valid_device) {
314       std::cout << "Failed to program any device found, exit!\n";
315       exit(EXIT_FAILURE);
316   }
317
318   // Allocate Buffer in Global Memory
319   // Buffers are allocated using CL_MEM_USE_HOST_PTR for efficient memory and
320   // Device-to-host communication
321
322   OCL_CHECK(err, cl::Buffer buffer_v1(context, CL_MEM_USE_HOST_PTR |
    ↪  CL_MEM_READ_ONLY, sizeof(TYPE) * NNZ,
323                                       source_val.data(), &err));
324   OCL_CHECK(err, cl::Buffer buffer_v2(context, CL_MEM_USE_HOST_PTR |
    ↪  CL_MEM_READ_ONLY, sizeof(int32_t) * NNZ,
325                                       source_cols.data(), &err));
326   OCL_CHECK(err, cl::Buffer buffer_v3(context, CL_MEM_USE_HOST_PTR |
    ↪  CL_MEM_READ_ONLY, sizeof(int32_t) * (N+1),
327                                       source_rowDelimiters.data(), &err));
328   OCL_CHECK(err, cl::Buffer buffer_v4(context, CL_MEM_USE_HOST_PTR |
    ↪  CL_MEM_READ_ONLY, sizeof(TYPE) * N,
329                                       source_vec.data(), &err));
330
331   OCL_CHECK(err, cl::Buffer buffer_v5(context, CL_MEM_USE_HOST_PTR |
    ↪  CL_MEM_WRITE_ONLY, sizeof(TYPE) * N,
332                                       target_out.data(), &err));
333
334   /* Default version */
335   OCL_CHECK(err, err = krnl_add.setArg(0, buffer_v1));
336   OCL_CHECK(err, err = krnl_add.setArg(1, buffer_v2));
337   OCL_CHECK(err, err = krnl_add.setArg(2, buffer_v3));
338   OCL_CHECK(err, err = krnl_add.setArg(3, buffer_v4));
339   OCL_CHECK(err, err = krnl_add.setArg(4, buffer_v5));
340
341   // Copy input data to device global memory
```

```
342   OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_v1, buffer_v2, buffer_v3,
      ↪  buffer_v4}, 0 /* 0 means from host*/));
343   OCL_CHECK(err, err = q.finish());
344
345   // Launch the Kernel
346   // For HLS kernels global and local size is always (1,1,1). So, it is recommended
347   // to always use enqueueTask() for invoking HLS kernel
348   std::cout << "Starting kernel\n";
349   OCL_CHECK(err, err = q.enqueueTask(krnl_add));
350   std::cout << "Waiting for accelerator to finish\n";
351   OCL_CHECK(err, err = q.finish());
352   std::cout << "Waiting for accelerator to finish2\n";
353   // Copy Result from Device Global Memory to Host Local Memory
354   //OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_aligna, buffer_alignb,
      ↪  buffer_m, buffer_ptr}, CL_MIGRATE_MEM_OBJECT_HOST));
355   OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_v5},
      ↪  CL_MIGRATE_MEM_OBJECT_HOST));
356   std::cout << "Waiting for accelerator to finish3\n";
357   OCL_CHECK(err, err = q.finish());
358   std::cout << "Waiting for accelerator to finish4\n";
359   // OPENCL HOST CODE AREA END
360
361   auto stop = high_resolution_clock::now();
362   auto duration = duration_cast<microseconds>(stop - start);
363
364   std::cout << "Execution time: " << duration.count() << std::endl;
365
366   printf("Fetching data from accelerator %f\n", target_out[0]);
367
368
369   for (int i=0; i < NNZ; i++) {
370     args->val[i] = source_val[i];
371     args->cols[i] = source_cols[i];
372   }
373   for (int i=0; i < (N+1); i++) {
374     args->rowDelimiters[i] = source_rowDelimiters[i];
375   }
376   for (int i=0; i < N; i++) {
377     args->vec[i] = source_vec[i];
378     args->out[i] = target_out[i];
379   }
380
381   /*for (int i=0; i < ALEN; i++) {
382     args->seqA[i] = source_seqa[i];
383   }
384   printf("Data fetched02\n");
385   for (int i=0; i < BLEN; i++) {
```

```
386       args->seqB[i] = source_seqb[i];
387     }
388     printf("Data fetched03\n");
389     for (int i=0; i < (ALEN+BLEN); i++) {
390       args->alignedA[i] = target_aligna[i];
391       args->alignedB[i] = target_alignb[i];
392     }*/
393     /*printf("Data fetched04\n");
394     for (int i=0; i < ((ALEN+1)*(BLEN+1)); i++) {
395       args->M[i] = rw_m[i];
396       args->ptr[i] = rw_ptr[i];
397     }*/
398
399     //printf("%f ?= %f\r\n", args->m1[0], args->prod[0]);
400     printf("Data fetched10\n");
401
402
403
404
405     // Parse command line.
406     const char *check_file = "data/check.data";
407
408     char *data = (char*)vargs;
409
410     // Load check data
411     printf("Checking output\n");
412     int check_fd;
413     char *ref;
414     ref = (char*) malloc(INPUT_SIZE);
415     assert( ref!=NULL && "Out of memory" );
416     check_fd = open( check_file, O_RDONLY );
417     assert( check_fd>0 && "Couldn't open check data file");
418     output_to_data(check_fd, ref);
419
420     // Validate benchmark results
421     printf("Validating output\n");
422     if( !check_data(data, ref) ) {
423       fprintf(stderr, "Benchmark results are incorrect\n");
424       //return -1;
425     } else {
426       fprintf(stderr, "BENCH SUCCESS!\n");
427     }
428     printf("Free!\n");
429     //free(data);
430     free(ref);
431
432     printf("Success.\n");
```

```
433   //exit(0);
434   //run_benchmark1(vargs);
435   if (ex) {
436     run_benchmark_gen(vargs, false);
437   }
438   exit(0);
439 }
440
441
442 void run_benchmark( void *vargs ) {
443   //run_benchmark_gen(vargs, false);
444   run_benchmark_daer(vargs, false);
445 }
446
447 /* Input format:
448 %% Section 1
449 TYPE[NNZ]: the nonzeros of the matrix
450 %% Section 2
451 int32_t[NNZ]: the column index of the nonzeros
452 %% Section 3
453 int32_t[N+1]: the start of each row of nonzeros
454 %% Section 4
455 TYPE[N]: the dense vector
456 */
457
458 void input_to_data(int fd, void *vdata) {
459   struct bench_args_t *data = (struct bench_args_t *)vdata;
460   char *p, *s;
461   // Zero-out everything.
462   memset(vdata,0,sizeof(struct bench_args_t));
463   // Load input string
464   p = readfile(fd);
465
466   s = find_section_start(p,1);
467   STAC(parse_,TYPE,_array)(s, data->val, NNZ);
468
469   s = find_section_start(p,2);
470   parse_int32_t_array(s, data->cols, NNZ);
471
472   s = find_section_start(p,3);
473   parse_int32_t_array(s, data->rowDelimiters, N+1);
474
475   s = find_section_start(p,4);
476   STAC(parse_,TYPE,_array)(s, data->vec, N);
477   free(p);
478 }
479
```

```
480 void data_to_input(int fd, void *vdata) {
481   struct bench_args_t *data = (struct bench_args_t *)vdata;
482
483   write_section_header(fd);
484   STAC(write_,TYPE,_array)(fd, data->val, NNZ);
485
486   write_section_header(fd);
487   write_int32_t_array(fd, data->cols, NNZ);
488
489   write_section_header(fd);
490   write_int32_t_array(fd, data->rowDelimiters, N+1);
491
492   write_section_header(fd);
493   STAC(write_,TYPE,_array)(fd, data->vec, N);
494 }
495
496 /* Output format:
497 %% Section 1
498 TYPE[N]: The output vector
499 */
500
501 void output_to_data(int fd, void *vdata) {
502   struct bench_args_t *data = (struct bench_args_t *)vdata;
503   char *p, *s;
504   // Load input string
505   p = readfile(fd);
506
507   s = find_section_start(p,1);
508   STAC(parse_,TYPE,_array)(s, data->out, N);
509   free(p);
510 }
511
512 void data_to_output(int fd, void *vdata) {
513   struct bench_args_t *data = (struct bench_args_t *)vdata;
514
515   write_section_header(fd);
516   STAC(write_,TYPE,_array)(fd, data->out, N);
517 }
518
519 int check_data( void *vdata, void *vref ) {
520   struct bench_args_t *data = (struct bench_args_t *)vdata;
521   struct bench_args_t *ref = (struct bench_args_t *)vref;
522   int has_errors = 0;
523   int i;
524   TYPE diff;
525   //printf("%f ?= %f\n", data->out[0], ref->out[0]);
526
```

```
527   for(i=0; i<N; i++) {
528     diff = data->out[i] - ref->out[i];
529     has_errors |= (diff<-EPSILON) || (EPSILON<diff);
530     //printf("%d, %d %f?=%f\n", i, has_errors, data->out[i], ref->out[i]);
531   }
532
533   // Return true if it's correct.
534   return !has_errors;
535 }
```

Listing B.7: spmv: Host code

B.3 stencil2d

```
1  #include "stencil.h"
2
3  void stencil(TYPE *orig, TYPE *sol, TYPE *filter){
4  #pragma HLS INTERFACE s_axilite port=orig
5  #pragma HLS INTERFACE s_axilite port=sol
6  #pragma HLS INTERFACE s_axilite port=filter
7  #pragma HLS INTERFACE s_axilite port=return
8  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 port=orig bundle=gmem0
9  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 port=sol bundle=gmem1
10 #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 port=filter
   ↪ bundle=gmem2
11     int r, c, k1, k2;
12     TYPE temp, mul;
13
14     stencil_label1:for (r=0; r<row_size-2; r++) {
15 //#pragma HLS PIPELINE
16         stencil_label2:for (c=0; c<col_size-2; c++) {
17 #pragma HLS PIPELINE
18             temp = (TYPE)0;
19             stencil_label3:for (k1=0;k1<3;k1++){
20 #pragma HLS PIPELINE
21                 stencil_label4:for (k2=0;k2<3;k2++){
22 #pragma HLS PIPELINE
23                     mul = filter[k1*3 + k2] * orig[(r+k1)*col_size + c+k2];
24                     temp += mul;
25                 }
26             }
27             sol[(r*col_size) + c] = temp;
28         }
29     }
30 }
```

Listing B.8: stencil2d: Kernel original code

```
#include <stdlib.h>
#include <stdio.h>
#include <hls_stream.h>
#include "./stencil.h"
void fetch_unit0(int32_t * filter, hls::stream<int32_t> & filter_rs0) {
  int r;
  int c;
  int k1;
  int k2;
  fetch_unit0_stencil_label1: for(r = 0; r<row_size - 2; r ++) {
//#pragma HLS PIPELINE
    fetch_unit0_stencil_label2: for(c = 0; c<col_size - 2; c ++) {
#pragma HLS PIPELINE
      fetch_unit0_stencil_label3: for(k1 = 0; k1<3; k1 ++) {
#pragma HLS PIPELINE
        fetch_unit0_stencil_label4: for(k2 = 0; k2<3; k2 ++) {
#pragma HLS PIPELINE
          filter_rs0.write(filter[k1 * 3 + k2]);
        }
      }
    }
  }
}
void fetch_unit1(int32_t * orig, hls::stream<int32_t> & orig_rs1) {
  int r;
  int c;
  int k1;
  int k2;
  fetch_unit1_stencil_label1: for(r = 0; r<row_size - 2; r ++) {
//#pragma HLS PIPELINE
    fetch_unit1_stencil_label2: for(c = 0; c<col_size - 2; c ++) {
#pragma HLS PIPELINE
      fetch_unit1_stencil_label3: for(k1 = 0; k1<3; k1 ++) {
#pragma HLS PIPELINE
        fetch_unit1_stencil_label4: for(k2 = 0; k2<3; k2 ++) {
#pragma HLS PIPELINE
          orig_rs1.write(orig[(r + k1) * col_size + c + k2]);
        }
      }
    }
  }
}
void write_unit2(int32_t * sol, hls::stream<int32_t> & sol_ws2) {
  int r;
  int c;
  int32_t temp;
```

```
47    int k1;
48    int k2;
49    int32_t mul;
50    write_unit2_stencil_label1: for(r = 0; r<row_size - 2; r ++) {
51  #pragma HLS PIPELINE
52      write_unit2_stencil_label2: for(c = 0; c<col_size - 2; c ++) {
53  #pragma HLS PIPELINE
54        sol[(r * col_size) + c] = sol_ws2.read();
55      }
56    }
57  }
58  void process_unit(hls::stream<int32_t> & filter_rs0, hls::stream<int32_t> &
    ↪  orig_rs1,
59                    hls::stream<int32_t> & sol_ws2) {
60    int r;
61    int c;
62    int32_t temp;
63    int k1;
64    int k2;
65    int32_t mul;
66    stencil_label1: for(r = 0; r<row_size - 2; r ++) {
67  //#pragma HLS PIPELINE
68      stencil_label2: for(c = 0; c<col_size - 2; c ++) {
69  #pragma HLS PIPELINE
70        temp =(TYPE) 0;
71        stencil_label3: for(k1 = 0; k1<3; k1 ++) {
72  #pragma HLS PIPELINE
73          stencil_label4: for(k2 = 0; k2<3; k2 ++) {
74  #pragma HLS PIPELINE
75            mul = filter_rs0.read() * orig_rs1.read();
76            temp += mul;
77          }
78        }
79        sol_ws2.write( temp);
80      }
81    }
82  }
83  void stencil(int32_t * orig, int32_t * sol, int32_t * filter) {
84  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem0
    ↪  port=filter
85  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem1 port=orig
86  #pragma HLS INTERFACE m_axi max_widen_bitwidth=512 depth=100 bundle=gmem2 port=sol
87    hls::stream<int32_t> filter_rs0;
88  #pragma HLS STREAM variable=filter_rs0
89    hls::stream<int32_t> orig_rs1;
90  #pragma HLS STREAM variable=orig_rs1
91    hls::stream<int32_t> sol_ws2;
```

```
92 #pragma HLS STREAM variable=sol_ws2
93 #pragma HLS DATAFLOW
94   fetch_unit0(filter, filter_rs0);
95   fetch_unit1(orig, orig_rs1);
96   process_unit(filter_rs0, orig_rs1, sol_ws2);
97   write_unit2(sol, sol_ws2);
98 }
```

Listing B.9: stencil2d: Kernel translated code

```
1  #include "stencil.h"
2  #include <string.h>
3  #include "xcl2.hpp"
4  #include <vector>
5
6  #include <string.h>
7  #include <unistd.h>
8  #include <fcntl.h>
9  #include <sys/stat.h>
10 #include <assert.h>
11
12 #include <chrono>
13 using namespace std::chrono;
14
15 int INPUT_SIZE = sizeof(struct bench_args_t);
16
17 std::string binaryFile;
18
19 #define EPSILON ((TYPE)1.0e-6)
20
21 void run_benchmark( void *vargs ) {
22   struct bench_args_t *args = (struct bench_args_t *)vargs;
23   //stencil( args->orig, args->sol, args->filter );
24   printf("Preparing accelerator\n");
25   cl_int err;
26   cl::Kernel krnl_add;
27   cl::CommandQueue q;
28   cl::Context context;
29
30   /*
31     TYPE orig[row_size*col_size];
32     TYPE sol[row_size*col_size];
33     TYPE filter[f_size];
34   */
35   std::vector<TYPE, aligned_allocator<TYPE> > orig(row_size*col_size);
36   std::vector<TYPE, aligned_allocator<TYPE> > sol(row_size*col_size);
37   std::vector<TYPE, aligned_allocator<TYPE> > filter(f_size);
```

```
38
39    for (int i=0; i <(row_size*col_size); i++) {
40      orig[i] = args->orig[i];
41      sol[i] = args->sol[i];
42    }
43    for (int i=0; i < f_size; i++) {
44      filter[i] = args->filter[i];
45    }
46
47
48
49    //gemm( args->m1, args->m2, args->prod );
50    // OPENCL HOST CODE AREA START
51    // get_xil_devices() is a utility API which will find the xilinx
52    // platforms and will return list of devices connected to Xilinx platform
53    auto devices = xcl::get_xil_devices();
54    // read_binary_file() is a utility API which will load the binaryFile
55    // and will return the pointer to file buffer.
56    auto fileBuf = xcl::read_binary_file(binaryFile);
57    cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
58    bool valid_device = false;
59    for (unsigned int i = 0; i < devices.size(); i++) {
60        auto device = devices[i];
61        // Creating Context and Command Queue for selected Device
62        OCL_CHECK(err, context = cl::Context(device, nullptr, nullptr, nullptr,
      &err));
63        OCL_CHECK(err, q = cl::CommandQueue(context, device,
      CL_QUEUE_PROFILING_ENABLE, &err));
64        std::cout << "Trying to program device[" << i << "]: " <<
      device.getInfo<CL_DEVICE_NAME>() << std::endl;
65        cl::Program program(context, {device}, bins, nullptr, &err);
66        if (err != CL_SUCCESS) {
67            std::cout << "Failed to program device[" << i << "] with xclbin file!\n";
68        } else {
69            std::cout << "Device[" << i << "]: program successful!\n";
70            OCL_CHECK(err, krnl_add = cl::Kernel(program, "stencil", &err));
71            valid_device = true;
72            break; // we break because we found a valid device
73        }
74    }
75    if (!valid_device) {
76        std::cout << "Failed to program any device found, exit!\n";
77        exit(EXIT_FAILURE);
78    }
79
80    // Allocate Buffer in Global Memory
81    // Buffers are allocated using CL_MEM_USE_HOST_PTR for efficient memory and
```

```
82    // Device-to-host communication
83
84    OCL_CHECK(err, cl::Buffer buffer_i1(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(TYPE) * (row_size*col_size),
85                                        orig.data(), &err));
86    OCL_CHECK(err, cl::Buffer buffer_i2(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_READ_ONLY, sizeof(TYPE) * f_size,
87                                        filter.data(), &err));
88
89    OCL_CHECK(err, cl::Buffer buffer_o3(context, CL_MEM_USE_HOST_PTR |
      ↪  CL_MEM_WRITE_ONLY, sizeof(TYPE) * (row_size*col_size),
90                                        sol.data(), &err));
91
92    /* Default version */
93    OCL_CHECK(err, err = krnl_add.setArg(0, buffer_i1));
94    OCL_CHECK(err, err = krnl_add.setArg(1, buffer_o3));
95    OCL_CHECK(err, err = krnl_add.setArg(2, buffer_i2));
96
97    // Copy input data to device global memory
98    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_i1, buffer_i2}, 0 /* 0
      ↪  means from host*/));
99    OCL_CHECK(err, err = q.finish());
100
101   // Launch the Kernel
102   // For HLS kernels global and local size is always (1,1,1). So, it is
103   // recommended
104   // to always use enqueueTask() for invoking HLS kernel
105   std::cout << "Starting kernel\n";
106   auto start = high_resolution_clock::now();
107   OCL_CHECK(err, err = q.enqueueTask(krnl_add));
108   std::cout << "Waiting for accelerator to finish\n";
109   OCL_CHECK(err, err = q.finish());
110   std::cout << "Waiting for accelerator to finish2\n";
111   // Copy Result from Device Global Memory to Host Local Memory
112   //OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_aligna, buffer_alignb,
      ↪  buffer_m, buffer_ptr}, CL_MIGRATE_MEM_OBJECT_HOST));
113   OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_o3},
      ↪  CL_MIGRATE_MEM_OBJECT_HOST));
114   std::cout << "Waiting for accelerator to finish3\n";
115   OCL_CHECK(err, err = q.finish());
116   std::cout << "Waiting for accelerator to finish4\n";
117   // OPENCL HOST CODE AREA END
118
119   auto stop = high_resolution_clock::now();
120   auto duration = duration_cast<microseconds>(stop - start);
121
122   std::cout << "Execution time: " << duration.count() << std::endl;
```

```
123
124   printf("Fetching data from accelerator\n");
125
126
127   //printf("Data fetched01\n");
128
129   for (int i=0; i <(row_size*col_size); i++) {
130     args->orig[i] = orig[i];
131     args->sol[i] = sol[i];
132   }
133   for (int i=0; i < f_size; i++) {
134     args->filter[i] = filter[i];
135   }
136
137   //printf("%f ?= %f\r\n", args->m1[0], args->prod[0]);
138   printf("Data fetched10\n");
139
140
141
142
143   // Parse command line.
144   const char *check_file = "data/check.data";
145
146   char *data = (char*)vargs;
147
148   // Load check data
149   printf("Checking output\n");
150   int check_fd;
151   char *ref;
152   ref = (char*) malloc(INPUT_SIZE);
153   assert( ref!=NULL && "Out of memory" );
154   check_fd = open( check_file, O_RDONLY );
155   assert( check_fd>0 && "Couldn't open check data file");
156   output_to_data(check_fd, ref);
157
158   // Validate benchmark results
159   printf("Validating output\n");
160   if( !check_data(data, ref) ) {
161     fprintf(stderr, "Benchmark results are incorrect\n");
162     //return -1;
163   } else {
164     fprintf(stderr, "BENCH SUCCESS!\n");
165   }
166   printf("Free!\n");
167   //free(data);
168   free(ref);
169
```

```
170   printf("Success.\n");
171   exit(0);
172
173 }
174
175 /* Input format:
176 %% Section 1
177 TYPE[row_size*col_size]: input matrix
178 %% Section 2
179 TYPE[f_size]: filter coefficients
180 */
181
182 void input_to_data(int fd, void *vdata) {
183   struct bench_args_t *data = (struct bench_args_t *)vdata;
184   char *p, *s;
185   // Zero-out everything.
186   memset(vdata,0,sizeof(struct bench_args_t));
187   // Load input string
188   p = readfile(fd);
189
190   s = find_section_start(p,1);
191   STAC(parse_,TYPE,_array)(s, data->orig, row_size*col_size);
192
193   s = find_section_start(p,2);
194   STAC(parse_,TYPE,_array)(s, data->filter, f_size);
195   free(p);
196 }
197
198 void data_to_input(int fd, void *vdata) {
199   struct bench_args_t *data = (struct bench_args_t *)vdata;
200
201   write_section_header(fd);
202   STAC(write_,TYPE,_array)(fd, data->orig, row_size*col_size);
203
204   write_section_header(fd);
205   STAC(write_,TYPE,_array)(fd, data->filter, f_size);
206 }
207
208 /* Output format:
209 %% Section 1
210 TYPE[row_size*col_size]: solution matrix
211 */
212
213 void output_to_data(int fd, void *vdata) {
214   struct bench_args_t *data = (struct bench_args_t *)vdata;
215   char *p, *s;
216   // Zero-out everything.
```

```
217   memset(vdata,0,sizeof(struct bench_args_t));
218   // Load input string
219   p = readfile(fd);
220
221   s = find_section_start(p,1);
222   STAC(parse_,TYPE,_array)(s, data->sol, row_size*col_size);
223   free(p);
224 }
225
226 void data_to_output(int fd, void *vdata) {
227   struct bench_args_t *data = (struct bench_args_t *)vdata;
228
229   write_section_header(fd);
230   STAC(write_,TYPE,_array)(fd, data->sol, row_size*col_size);
231 }
232
233 int check_data( void *vdata, void *vref ) {
234   struct bench_args_t *data = (struct bench_args_t *)vdata;
235   struct bench_args_t *ref = (struct bench_args_t *)vref;
236   int has_errors = 0;
237   int row, col;
238   TYPE diff;
239
240   for(row=0; row<row_size; row++) {
241     for(col=0; col<col_size; col++) {
242       diff = data->sol[row*col_size + col] - ref->sol[row*col_size + col];
243       has_errors |= (diff<-EPSILON) || (EPSILON<diff);
244     }
245   }
246
247   // Return true if it's correct.
248   return !has_errors;
249 }
```

Listing B.10: stencil2d: Host code