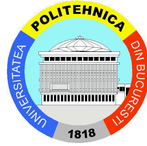
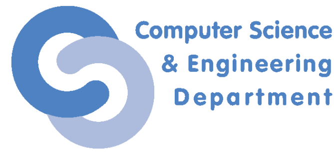
Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare





LUCRARE DE DISERTAȚIE

Compilator pentru procesare vectorială și scalarea frecvenței la nivelul instrucțiunilor

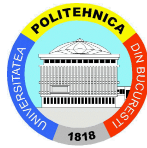
Coordonatori Științifici:
ș.l. dr. ing. Lucian Petrică
ș.l. dr. ing. Adrian-Răzvan Deaconescu

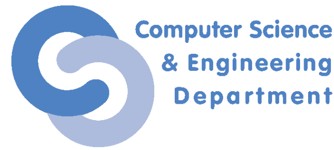
Autor:
Laura Mihaela Vasilescu

București, Septembrie 2014

University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department





MASTER THESIS

Compiler for Vector Processing and Frequency Scaling at Instruction Level

Scientific Advisers:
ș.l. dr. ing. Lucian Petrică
ș.l. dr. ing. Adrian-Răzvan Deaconescu

Author:
Laura Mihaela Vasilescu

Bucharest, September 2014

Row, row, row your boat,

Gently down the stream.

Merrily, merrily, merrily, merrily,

Life is but a dream.

Abstract

Computer vision is a prominent research topic of AI (Artificial Intelligence) that aims to build agents capable of recognizing objects, movements, patterns and extract information in the same way as human vision or even better.

Software performance is often limited by the hardware underneath. To increase the overall performance, the current research trend is to offload some of the computation to a dedicated processor. This thesis focuses on designing and implementing a modular and without language constrains compiler for the ConnexArray architecture and introducing awareness of the frequency scaling at instruction level of the VASILE architecture.

Keywords: *compiler, FPGA, LLVM, OpenCL*

List of Abbreviations

AI Artificial Intelligence

CPU Central Processing Unit

CV Computer Vision

DMA Direct Memory Access

EU Execution Unit

FPGA Field-Programmable Gate Array

FSU Frequency Selection Unit

GCC GNU Compiler Collection

GPU Graphics Processing Unit

IFDDU Instruction Fetch Decode and Dispatch Unit

IPS Instructions Per Second

IR Intermediate Representation

ISA Instruction Set Architecture

LLVM Low Level Virtual Machine

LLVM IR LLVM Intermediate Representation

MIMD Multiple Instruction, Multiple Data

OpenCL Open Computing Language

OpenCV Open Source Computer Vision

OPINCAA OPcode INjector for the ConnexArray Architecture

POCL POrtable Computing Language

SIMD Single Instruction, Multiple Data

SSA Static-Single Assignment

VASILE Vector Architecture for frequency Scaling at Instruction LEvel

Contents

Lullabies i

Abstract ii

1 Introduction 1
- 1.1 Motivation: Computer Vision Applications . . . 1
- 1.2 Resource Requirements for Computer Vision Applications . . . 3
- 1.3 Software vs. Hardware Optimizations . . . 4
  - 1.3.1 Software Optimizations . . . 4
  - 1.3.2 Hardware Optimizations . . . 5

2 Project Overview 7
- 2.1 ConnexArray Architecture . . . 7
- 2.2 VASILE Architecture . . . 10
- 2.3 Compiler: Tools of the Trade . . . 12
  - 2.3.1 LLVM Compiler Infrastructure . . . 12
  - 2.3.2 POCL: Open Source Runtime Library for OpenCL . . . 13
  - 2.3.3 OPINCAA . . . 14

3 Compiler Infrastructure 15
- 3.1 Host Compiler . . . 16
- 3.2 Accelerator Compiler . . . 17

4 Runtime Library 18
- 4.1 Implementation . . . 18
- 4.2 Memory Mapping Model . . . 20
- 4.3 Data Transfers . . . 21
- 4.4 Instruction Fetching . . . 22
- 4.5 Validation Results . . . 22
  - 4.5.1 Accuracy of Data Transfers . . . 22
  - 4.5.2 Accuracy of Instruction Fetching . . . 23

5 Backend: Code Generator 24
- 5.1 Generation of LLVM IR . . . 24
- 5.2 Generation of ConnexArray Machine Code . . . 25
  - 5.2.1 Code Generation Design . . . 25
  - 5.2.2 Register Allocator . . . 27

5.3 Evaluation and Results . . . 27
5.3.1 Validation . . . 28
5.3.2 Evaluation . . . 29

6 Frequency Scaling at Instruction Level 30
6.1 Testing scenario . . . 30
6.2 Hardware Calibration Phase . . . 31
6.3 Loop Tiling Optimization . . . 32
6.4 Evaluation . . . 32

7 Conclusion and Future Development 34

A Instruction Set Architecture (ISA) for ConnexArray 35

B Simple Example of OpenCL Program 37

List of Figures

1.1 Relation between computer vision and various other fields [1] 2

2.1 Vector Processor Components 8
2.2 Layout of the Zynq FPGA silicon die (image from [14]) 10
2.3 VASILE Processing Element Structure 11

3.1 Host-Accelerator System 15
3.2 Compilation Process and Execution Flow 16

4.1 OpenCL Generic Memory Model [2] 20

5.1 Code Generation Diagram Flow 24
5.2 Backend Classes Diagram 26

List of Tables

1.1 Instructions Per Second (IPS) for different types of processors. 3

4.1 Correspondence between `OpenCL API` and `POCL` hooks 19

5.1 `getelementptr` Semantics 26
5.2 Bubble-Sort Execution Time on a Vector with 128,000 Elements 29

6.1 SSD Profiling Results 31
6.2 SSD Execution Time 33

A.1 Instruction Set Architecture for ConnexArray 36

List of Listings

2.1 Instruction Format with Immediate Value 9
2.2 Instruction Format without Immediate Value 10
2.3 OPINCAA Kernel Example 14
3.1 Host Machine Processor and Compilers Available 16
4.1 `struct pocl_device_ops` Initialization 18
5.1 OpenCL Kernel: Increment Values 25
5.2 LLVM IR Kernel: Increment Values 25
5.3 OpenCL Kernel: Modified Bubble Sort Algorithm 28
6.1 OpenCL Kernel: SSD Algorithm 31
6.2 Tiled SSD Algorithm 32
B.1 Simple Example of OpenCL Program 37

Chapter 1

Introduction

Computer vision is a prominent research topic of AI (Artificial Intelligence) that aims to build agents capable of recognizing objects, movements, patterns and extract information in the same way as human vision or even better. This project aims to simplify the development process of computer vision applications for ConnexArray and VASILE (Vector Architecture for frequency Scaling at Instruction LEvel) architectures.

ConnexArray is a SIMD processor implemented in FPGA that has 128 execution units. VASILE is an enhancement of the ConnexArray architecture that enables the adjustment of the frequency at instruction level. Both architectures are suitable for computer vision applications.

1.1 Motivation: Computer Vision Applications

Mankind was always preoccupied with creating tools and machineries in order to reduce the physical work and improve their life quality. One good example of such a machinery is the clock. People have always been concerned about measuring time. They were not doing this as a hobby, instead, they were forced to find a good way to predict seasons to improve the harvest and hunting. Ancient people used different methods and machineries to keep track of the time. They used obelisks[1] to track the movement of the Sun, water clocks[2] and, later, sand glasses[3]. Those systems were pretty big as size and, at that moment, no one thought about the possibility of having a small clock inside your pocket or, even more, the possibility to make the clock alert you in a few hours.

This is what we call AI (Artificial Intelligence). This research field began in 1956 as a research conference[4] where the attendees wrote programs that were solving mathematical problems. Nowadays, those kind of problems are considered basic problems and are not necessarily part of the AI field. The evolution of technology

[1] http://en.wikipedia.org/wiki/Obelisk
[2] http://en.wikipedia.org/wiki/Water_clock
[3] http://en.wikipedia.org/wiki/Hourglass
[4] http://en.wikipedia.org/wiki/Dartmouth_Conferences

was impressive during the past decades and machineries are integral part of our lives. We can barely imagine how our lives would look like without modern cars, computers, search engines, mobile applications. Our lives are greatly simplified by intelligent agents that correlate billions of data (big data).

At the beginning, every AI researcher's goal was to build a machine as intelligent as a human, but soon they realised the difficulty of the project. Today, AI agents are built to resolve specific problems and, in in most of the cases (like chess), they solve problems better than humans. Some of the specific problems resolved by AI agents simulate different humans abilities, like perception.

Perception is a human ability to receive and interpret information from the environment. They have many senses: *ophthalmoception* (vision), *audioception* (hearing), *gustaoception* (taste), *olfacception* (smell), *tactioception* (touch), *thermoception* (temperature), *proprioception* (kinesthetic), *nociception* (pain), etc.

Ophthalmoception (visual perception) is the ability to interpret information from the visible spectrum. In AI this type of problems are part of the **computer vision** field: navigation (autonomous cars or robots - drones), controlling processes (industrial robots), detecting events (surveillance), information labeling (databases of images), topographical modeling, medical image analysis, etc.

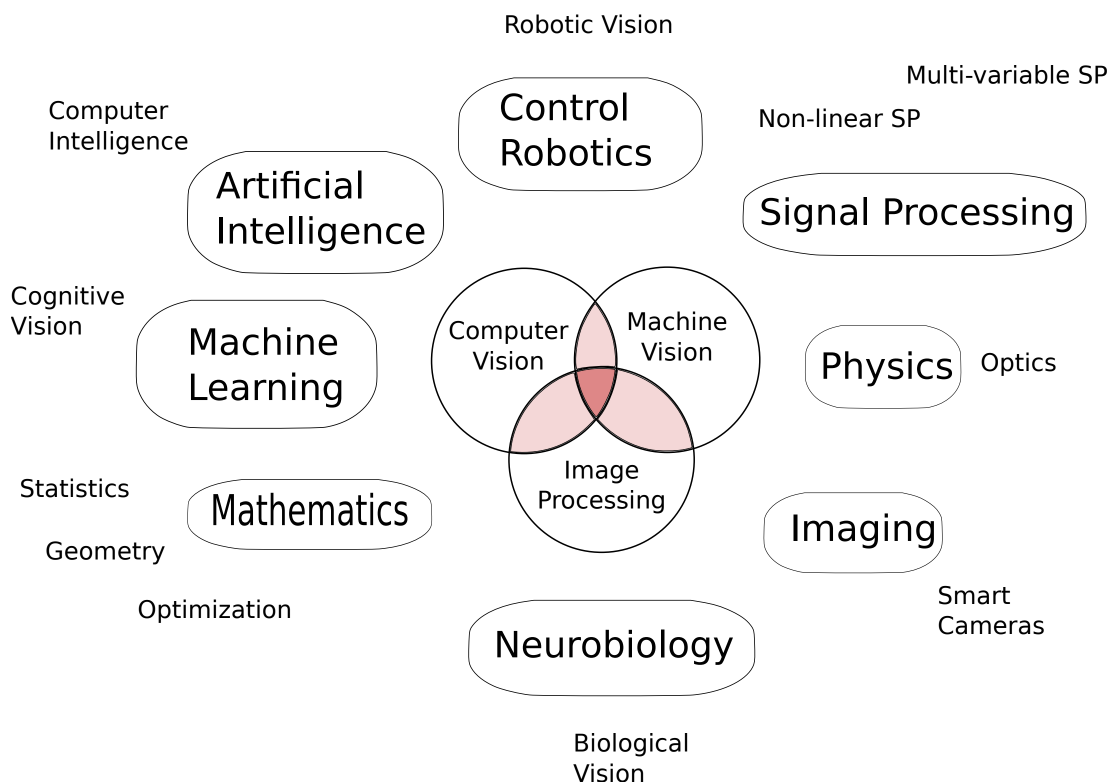


Figure 1.1: Relation between computer vision and various other fields [1]

Computer vision is not only about understanding visual images (both video and photography), but it is also about acquiring methods, data representation, mathematical

[1]image from http://en.wikipedia.org/wiki/Computer_vision (23.03.2014, 18:37)

models and representations and algorithms. It is a very complex field that interconnects several other scientific fields.

In Figure 1.1, one can see the relation between computer vision and various other fields. It combines mathematics, algorithms design, hardware design and biology. There are three sub-areas of study of computer vision: *computer vision* itself that combines sciences like artificial intelligence and machine learning, *machine vision* that combines sciences like robotics, signal processing and physics and *image processing* that focus more on mathematical models for neurobiology concepts. The three sub-areas cannot be studied without minimum knowledge of the other sub-areas. Researchers that study this field need to have an interdisciplinary background or at least interest in order to combine different techniques and specific results from one domain to another.

1.2 Resource Requirements for Computer Vision Applications

Computer vision is a highly computationally intensive field. For example, a small image of 800 pixels wide and 600 pixels high, contains 480,000 individual values (pixels). Each pixel is typically characterized by 3 different values (one for red, one for green and one for blue) that form the pixel color. In total, that means almost 1.5 million values. But computer vision greater goal is to emulate human vision functions, so if we multiply the prior value with 30 (the typical frame rate for videos) we will get 50 million values per second to compute. Computer vision algorithms imply complex calculus and for each value the processor must be able to compute even thousands of instructions per seconds.

There are different techniques to reduce the problem size: some of them narrow the frame size, while others focus on reducing the number of colors. Those techniques will gain a significant speedup because what they do is basically decreasing the data size. There is often a trade off that has to be established between the accuracy of the results and the time it takes to extract the results.

Table 1.1 shows the number of IPS (Instructions Per Second) for three different types of CPUs.

Year	Processor	Millions of IPS
1994	Intel Pentium	188
2011	Intel Core i7 875K	92,100
2011	ARM Cortex A7	2,850

Table 1.1: Instructions Per Second (IPS) for different types of processors.

The first CPU from the table (Intel Pentium) was produced 20 years ago and it was very common for personal use. The number of IPS was too little to compute even a very basic computer vision algorithm. Even if people had ideas and wanted to explore in the area of image recognition, hardware resources weren't enough to do this in real time.

The second CPU from the table (Intel Core i7) is a CPU from the latest generation

and it's also commonly used for personal use, and even for servers. It works 500 times better than the old Intel Pentium processor and it can be suitable for simple and common computer vision applications. Such a processor can be used for algorithms with no more than 1,800 instructions for each pixel (for the small image that we took as an example before). Complex applications need more resources, but today CPUs can perform well for common tasks.

But, as technology evolves, people want to include computer vision applications in small devices like smartphones (that run Android[1], iOS[2], etc.) or wearables (Google Glass[3], Oculus Rift[4], OrCam[5], etc.). Such devices use CPUs processors specially designed for embedded systems. The reason is that those devices need to have a balance between performance and energy consumption, since most of the time they are not plugged in a power socket. The last CPU from the table is such a processor. The number of IPS is quite small and shows that such a processor can be used for algorithms with no more than 50 instructions for each pixel. That means that real-time processing for computer vision applications can't be done by using such a device.

In order to satisfy user demands, researchers are still studying different methods on how to provide real-time computing for computer vision applications in small devices. This is also what the current research project aims for.

1.3 Software vs. Hardware Optimizations

There are different techniques addressed by computer vision researchers.

On one hand, computer scientists focus more on the algorithms and find new ways of acquiring, representing and processing data. Algorithm optimization research goes thoroughly intertwined with the evolution of mathematical models. These optimizations techniques are software optimizations.

On the other hand, electrical engineers focus more on providing hardware capable of real-time executions with minimum power consumption. These optimizations techniques are hardware optimizations.

1.3.1 Software Optimizations

Computer vision algorithms arise from numerical models. In order to improve an algorithm complexity and execution time, one has to combine different models and resolve complex calculus. One book that presents such techniques is *Optimization for Computer Vision* [20], which presenting various mathematical techniques and tips of how one can combine them to improve algorithms. It starts with basic continuous optimizations like coordinate ascent, descent directions, Newton's method, trust regions, dogleg methods, subspace methods, DFP, BFGS and limited memory methods. After that,

[1] http://www.android.com
[2] https://www.apple.com/ios
[3] http://www.google.com/glass/start
[4] http://www.oculusvr.com
[5] http://www.orcam.com

the book is presenting more advanced methods of constrained optimization methods: lagrangians, quadric penalty methods, logistic regression methods, stochastic gradient descent, homotopy paths, pursuit methods, quadratic programming.

It is somehow interesting how those advanced techniques are combined with classic techniques from algorithms and data structures. Any combinatorial optimization involve flows and cuts, similar with alpha-beta pruning. The pruning is required because, in the end, the hardware has limited capabilities. In an ideal scenario, the cuts wouldn't be necessary and the algorithm would always give a 100% accuracy without any time penalty.

This is the reason why any software optimization is limited by the underlying hardware and the reason why it is difficult to build a computer vision application that will run as fast on any hardware. Complex applications are, in general, designed for specific hardware and they are jointly delivered.

However, writing applications for a specific hardware makes them difficult to be ported if a new hardware is released. The general solution is to use a library as a front-end for the computer vision algorithms that support several different hardwares. Such a library, that is commonly used, is OpenCV [1]. The library is open source and had its first stable release in 2006.

1.3.2 Hardware Optimizations

Hardware optimizations are mechanisms of providing hardware capabilities of real-time execution with minimum consumption power.

There are several institutes and universities that conduct research in this area. The oldest research systems are:

- BiTEC[2] - a solution for education/research and low volume image processing and machine vision applications
- 3D Laser Scanner [3] - a solution that offers information on the mechanical hardware
- VLSI Vision Chips [4] - a solutions that offers summaries of VLSI sensors

Since then, almost every university has developed a research group that is preoccupied with computer vision topics. Google Scholar reports almost 3 million research papers published in conferences that are related with the computer vision topic (date: 15.08.2014).

Fung et al. explains in [4] the advantages of offloading the computational part to `GPU`s. The analogy is quite simple: they say that usually a `GPU` is used to transform "numbers into pictures" and their proposal is to use the `GPU` in reverse, to assist in "converting pictures into numbers". Their paper appeared soon after the release of `NVIDIA CUDA`[5]

[1] http://opencv.org
[2] http://www.bitec.ltd.uk
[3] http://www.muellerr.ch/engineering/laserscanner/default.htm
[4] http://www.visionchips.com
[5] http://www.nvidia.com/object/cuda_home_new.html

programming model, a model of expressing program parallelism without the need for graphics expertise, but, instead by taking advantage of the existing hardware and offloading computational parts from CPU to other units. They do not build specially design hardware.

Ratha et al. describes [15] why Von Neumann[1] architecture doesn't meet the requirements of computer vision algorithms. They define this algorithm's category as a intensive calculus problem, with very little data dependency, and explains why SIMD architectures are preferred rather than MIMD architectures. Based on their observations, they created a list of architectural requirements:

- computational characteristic:
  - low-level vision algorithms are suitable for SIMD architectures
  - high-level vision algorithms are suitable for MIMD architectures
- high bandwidth I/O
- resource allocation: no wasting
- load balancing and task scheduling
- fault tolerance

Optimizations can be done at each point from the above list, but in order to avoid having a very powerful system with only one slow component that is acting like a bottleneck for the entire system, one has to take in consideration all of the requirements when choosing between different hardware.

The current trend is to build custom processing units in FPGA or ASIC. This approach has the advantage of creating the hardware in such a way that will fit the software requirements of the application.

Because there are so many groups and the research topic is quite new, there is no standard hardware labeled as *the best*. Each hardware has its own benefits and works best in the scenario it was designed for.

Both hardware and software optimizations are useful and can help gaining significant speedup and accuracy in computer vision applications, but the weak link is the hardware. Improving hardware will automatically gain even more speedup in terms of software, because the constrains imposed on the software component can be relaxed.

Our work also focuses on improving hardware for computer vision applications. The current thesis describes the compilation techniques and it is structured as follows: chapter 2 describes the project overview: hardware architecture and useful tools, chapter 3 describes the compiler architecture. The following three chapters describe each of the compiler components: the runtime library (chapter 4), the code generator (chapter 5) and the algorithm for frequency scaling at instruction level optimization (chapter 6).

[1] http://en.wikipedia.org/wiki/Von_Neumann_architecture

Chapter 2

Project Overview

In our goal to create hardware optimizations for computer vision applications, we've employed the use of ConnexArray[13] (see section 2.1). ConnexArray is a SIMD processor implemented in FPGA that is able to scale to thousands of SIMD execution units. This processor has a very simple ISA suitable for mathematical computations and it was designed in this manner to avoid including too much logic inside the execution units. Being focused on mathematical computations, it makes it suitable for computer vision applications.

We wanted to run typical computer vision application on this hardware, but without a compiler, writing computer vision applications for ConnexArray meant writing everything from scratch in binary code. This approach was limitative and involved a tremendous effort and deep understanding of how the architecture works. Furthermore, typical open source applications couldn't be run on this architecture without manually translating the code to ConnexArray assembly.

As we aim to simplify the process of writing computer vision applications for ConnexArrat, one of the goal of this project is to provide a compiler for this architecture. Furthermore, another objective of the project is to design a method that will enhance the compiler to be aware of the capabilities of frequency scaling at instruction level. VASILE[14] (see section 2.2) is a derived architecture that has the ability to configure different frequencies (within some ranges) for each instruction. In chapter 6 we present the algorithm behind the instructions reordering that was designed in order to take full advantage of the hardware capabilities.

2.1 ConnexArray Architecture

The ConnexArray architecture is a SIMD architecture implemented in FPGA that is able to scale to thousands of SIMD execution units. Figure 2.1 shows the general schema of a vector processor: processing elements, local memory (data input-output), instruction fetch and dispatch unit, host processor and main memory. How are those components interconnected is up to the processor specification.

The current implementation of **ConnexArray** has 128 SIMD execution units and 2

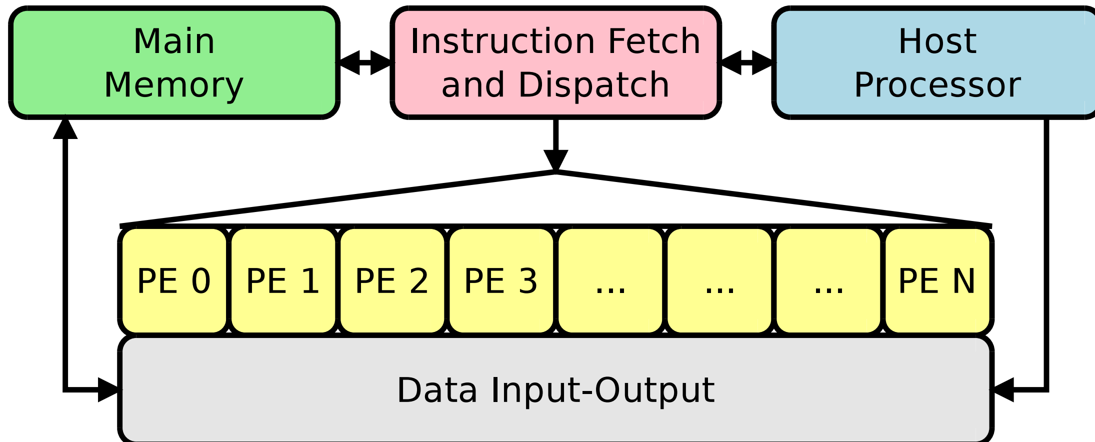


Figure 2.1: Vector Processor Components

kilobytes of local memory for each execution unit, but the ConnexArray has been previously used in commercial multimedia applications with 1024 execution units and various sizes of local memory [17][18]. Since the architecture is implemented in FPGA, it is very easy to reconfigure the entire architecture and to add various new extensions.

The ConnexArray consists of the following elements:

- *Controller*
  - handles instruction fetches and distributes them to the EUs
  - manages I/O, reduction and distribution operations
- *Execution Units (EU)*
  - mini-processors without memory subsystem that would normally fetch instructions and data from the main memory
  - contain a 16-word register file; each word is 16 bits wide
  - provide an interface to the local store (the only memory directly accessible by the EUs
  - support two addressing models: immediate and register-index
  - receive the same instruction simultaneously
- *Local Store*
  - private for each EU
  - 2KB size (1024 words)
  - has a single read-write port multiplexed between the EU and I/O Plane; the first one has priority
- *I/O Plane*
  - fetches data from main memory to the Local Store

  - transfers are done by the controller and use DMA transfer

- *Distribution Network*
  - used to distribute the same value to all EUs at the same time
  - implemented as a logarithmic tree which fans out a single 16-bit operand from the controller and writes it into the EUs register files

- *Reduction Network*
  - can be used to perform summation, bitwise OR, minimum and maximum calculations
  - implemented as a logarithmic tree of simple ALUs

- *Scan Network*
  - loops back over the array of EUs
  - helps with indexing operations

- *EU Interconnect*
  - gives the ability to have inter-communication between EUs
  - used in shift operations

The I/O system is coordinate by the central unit. The Local Store can be written and read through the I/O interface, which is of FIFO type (pipes). There are two FIFOs that forms the I/O interface:

- an inbound FIFO to transfer data from the host to the accelerator
- an outbound FIFO to transfer data from the accelerator to the host

Read and writes operations are from the perspective of the host. Each data transfer is initiated by a transfer descripto that occurs as follows: the host pushses the descriptor into the inbound FIFO, then, if the transfer is a write operations, the host pushes the write-data into the inbound FIFO and checks for transfer completion; if the transfer is a read operation, the hosts pops the read-data from the outbound FIFO. The sequences between the accelerator and the memory are transferred through DMA in parallel with the computational process.

Register addresses are 5 bits wide, therefore the architecture allows a maximum of 32 registers. The immediate value is 16 bits and requires the removal of the right operand address and the use of a reduced opcode. Therefore, there are 2 types of instructions formats as shown is listing 2.1 and listing 2.2.

```
-------------------------------------------------------------------
|   opcode   |       immediate value          |    left    |    dest    |
-------------------------------------------------------------------
|31(6 bits)26|25       (16 bits)            10|9 (5 bits) 5|4 (5 bits) 0|
-------------------------------------------------------------------
```

Listing 2.1: Instruction Format with Immediate Value

The opcode consists of a fixed section and a variable section which is formatted differently depending on the contents of the fixed section. Each opcode identifies uniquely a instruction. The switch between the instruction format is done by setting or unsetting the bit present at offset 7 of the opcode field (offset 30 of the instruction). If the bit is set (IMM=1), than the instruction format is the one with the immediate value.

```
-------------------------------------------------------------------
|    opcode      |  ---------  |    right    |    left    |   dest   |
-------------------------------------------------------------------
|31 (9 bits) 23|22(8 bits)15|14(5 bits)10|9 (5 bits) 5|4 (5 bits) 0|
-------------------------------------------------------------------
```

Listing 2.2: Instruction Format without Immediate Value

The entire instruction set of ConnexArray can be found in **Appendix A.1**.

2.2 VASILE Architecture

VASILE (**V**ector **A**rchitecture for **S**caling at **I**nstruction **LE**vel) [14] is a vector processor architecture implemented in FPGA technology that enables the adjustment of the frequency at instructions per second to enhance performance. It is a derived architecture from ConnexArray.

FPGA components vary in the mix of columns and their placement relative to each other. For VASILE we used a mid-range Zynq 7020[1] , who's silicon die layout can be seen in figure 2.2. The figure was extracted from the Xilinx PlanAhead[2] software and it was published in [14]. The maximum distance between embedded multiplier columns is half of the length of the die, which means the signal may travel up to a quarter of the die length to reach a multiplier. Memory columns are placed closer, so the signal travels to about 15% of the die length.

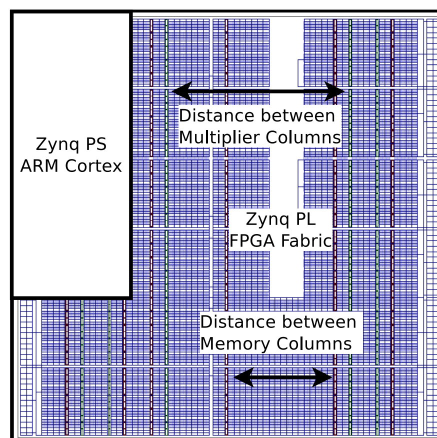


Figure 2.2: Layout of the Zynq FPGA silicon die (image from [14])

[1] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices/index.htm
[2] http://www.xilinx.com/tools/planahead.htm

The FPGA synthesis tool can place the logic close to the required memories and multipliers, but when a design occupies almost the FPGA's maximum capacity, some of the logic may not be placed close to the resources. `VASILE` overcomes this and enables an architectural solution to this problem.

`VASILE` segregates instructions based of their category in order to control data-path delays. The processing element organization is presented in figure 2.3. Its core consists of the register file, the associated flags, several operand registers, the forwarding logic and the write-back multiplexer (`MUX`). Each Instruction Unit (`IU`) receives input from the operand registers and connect its results to the `MUX`.

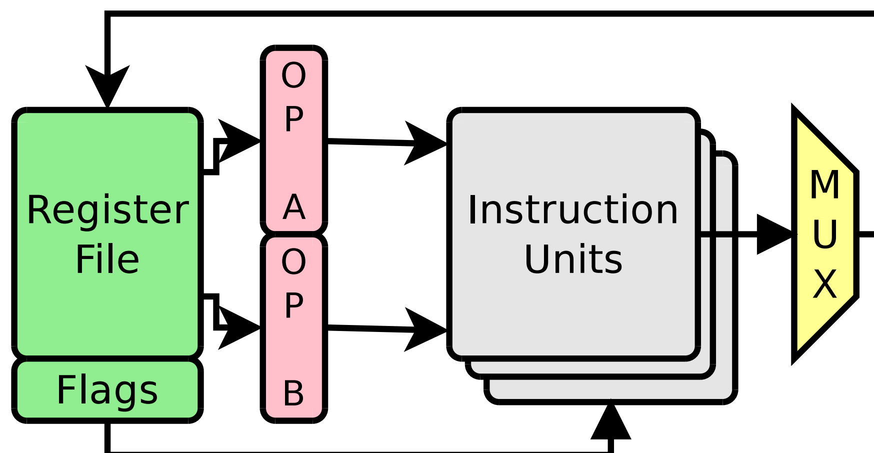


Figure 2.3: VASILE Processing Element Structure

Each class of instruction has a corresponding `IU` which utilize the same FPGA embedded resource. For example, algebra instructions are performed in `FPGA` logic (LUTs [22]). Multiplication are implemented using embedded multipliers. Memory access operations, like `load` and `store`, require access to embedded memory, while inter-communication instructions need a routing network to pass signals between `PE`s.

The decoding logic is performed in a central instruction fetch (`IFDDU`). This way, the `PE`s are simplified because they don't have integrated any decoding logic. The control signals are distributed through a pipelined logarithmic tree [16] to all `PE`s. The drawback is that the flip-flop utilisation is increased in the distribution tree, but the compromise is acceptable because the `PE`s don't use many `FPGA` flip-flops. For example, Xilinx `FPGA`s provide two flip-flops for each look-up table.

`VASILE` includes a **frequency selection unit (FSU)** to ensure different delays through the `IU`s. The `FSU` is formed by several clock sources, a clock `MUX` and the associated logic. Each clock source corresponds to one or more instruction classes. The instruction are inspected during the decode phase and than, the `FSU` selects the required clock source. The Instruction Fetch Decode and Dispatch Unit (`IFDDU`) ensures that the decoded instructions do not reach the `PE`s before the clock `MUX` switches to the required clock source for the instruction. This is done by adding a **configurable delay line**. The switch time is device-dependent and may require a relatively lengthly time to complete. However, the switching time is at most three slow clock periods:

$$T_{switch} <= 3 * max(T_1, T_2, ..., T_n)$$

In order to gain maximum performance for the VASILE architecture, a specialised compiler is required. To minimize the number of clock switches required, the compiler should tile loop and cluster instruction in blocks which belong to the same instruction class. This approach is discussed in chapter 6.

2.3 Compiler: Tools of the Trade

The biggest drawback for VASILE, in terms of performance, is the number of clock switches. The PEs are executing instructions sequential and make a switch every time the instruction class is changed. For example, every time a PE is executing a multiplication after an algebra instruction, a delay is introduced by the clock switching. In order to minimize the delays, the compiler should cluster instruction in blocks which belong to the same instruction class. [11]

Creating a compiler from scratch involves a lot of work and it can be a very limitative approach. A monolithic approach involves creating a compiler for a specific architecture and for only a specific programming language. If we were to choose such an approach, all the application developed for VASILE should been written in a specific programming language. But computer vision applications are written in many different programming languages, so, after summing up all the advantages and disadvantages, we choose to create a compiler using the LLVM[1] framework.

OpenCL[2][19] is a new industry standard for heterogeneous computing on a variety of modern CPUs, GPUs, DSPs and other microprocessor design [1]. This framework it is also widely used by the open source library for computer vision application (OpenCV[3]). Because of that, we decided to choose OpenCL as a front-end for our compiler.

This section is presenting the tools we choose to use in order to simplify our work.

2.3.1 LLVM Compiler Infrastructure

Low Level Virtual Machine (LLVM) [9] is an open source compiler infrastructure used for compile-time, link-time, run-time optimization of programs written in different programming languages. The initial purpose of the project was to create virtual machines for code execution, similar with Java Virtual Machine (JVM), but the project grew and now includes a variety of other compiler and low-level tools.

Not only that LLVM has the capability to transform the source code to a intermediate representation (LLVM IR), but can also accept the IR from other compilers, like GCC toolchain, and emit optimised IR. This way, LLVM can be used in cohesion with a wide variety of existing compilers.

The LLVM target-independent code generator is a framework under the umbrella of LLVM that has several components capable of translating LLVM IR to machine code

[1] http://llvm.org
[2] https://www.khronos.org/opencl
[3] http://opencv.org

for a specific target in binary or in assembly form. LLVM represents the instruction in their most abstract form to fulfill the target agnostic criteria.

LLVM code generator can represent sequence of instructions as machine instruction bundles.[12] Such a bundle can contain a variable number of parallel instructions or a sequential list of instructions that can not be separated (e.g. have data dependencies, etc.). A machine instruction bundle can also contain other machines instruction nested within.

By using LLVM, the VASILE compiler will need to implement a back-end compiler that will generate code from the LLVM IR. The advantage is that the top application can be written in any programming language supported by LLVM. Furthermore, this allows modifying and applying different kind of optimizations that can be architecture dependent or agnostic, like loop-unrolling, constant-propagation, etc.

2.3.2 POCL: Open Source Runtime Library for OpenCL

OpenCL is the first open standard for cross-platform, parallel programming of modern processors found in personal computers, servers and embedded devices. The standard only provides[1] an API definition, data representation format and the behavior of each of its function. Each hardware manufacture implements the standard through an SDK that is usually closed source (at least for the moment). For example, Intel's [2] SDK can be run only on their hardware and can't be used in conjunction with other vendors architectures. AMD[3] and NVIDIA[4] SDK's include other vendors SDKs to allow the developers to use their software and architecture in conjunction with other architectures as well. This approach gives less constrains and more flexibility for the developers.

The standard is open and architecture independent. It uses a C API on the host machine and a separate high-level kernel language for code definition. The kernel is the code that will run on the heterogeneous computers. The language used in the kernels has explicit support for vectorization so it is highly recommended for SIMD architectures. By using OpenCL, the programmer must write separate code for the host and accelerator, which can be handled by different compilers.

POrtable Computing Language[5] (POCL) is an open source implementation of the OpenCL standard that can easily be adopted for new targets and devices, both for homogeneous CPUs and heterogeneous GPUs/accelerators. It's not the only open source implementation, but it is the most advanced and also receives support from other companies, like Nokia[6].

We choose to use POCL not only of its advanced developing stage, but also because it uses clang[7], which is the compiler provided by LLVM for C applications.

[1] http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf
[2] https://software.intel.com/en-us/vcsource/tools/opencl-sdk
[3] http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk
[4] https://developer.nvidia.com/opencl
[5] http://portablecl.org
[6] https://research.nokia.com/cognitive_radio
[7] http://clang.llvm.org

2.3.3 OPINCAA

`OPcode INjector and Control for Accelerator Architectures` (`OPINCAA`) [2] is a just-in-time source-to-source translator that allows the user to define kernels in a human readable assembly syntax for `ConnexArray`. What it basically does is to deassemble the instructions from a text version into machine binary format.

Listing 2.3 shows an example of how the kernels look like when are written using `OPINCAA` library. The kernel does a shift to right operation for the first location of the Local Store and compare the value received from the left cell with the second value of the Local Store. The result is a sum of the indexes in which the two values where equal. The kernel lines of code are almost assembly language, but the representation if human friendly and the programmers can avoid writting directly binary code (see Appendix A.1).

```
1   _BEGIN_KERNEL("shift_right");
2       EXECUTE_IN_ALL(
3                       R1 = INDEX;
4                       R2 = LS[0];
5                       CELL_SHR(R1,R2);
6                       R3 = 0;
7                       R5 = LS[1];
8                       NOP;
9                       R4 = SHIFT_REG;
10                      R6 = (R5 == R4);
11                      NOP;
12                      )
13      EXECUTE_WHERE_EQ(R3 = INDEX;)
14      EXECUTE_IN_ALL( REDUCE(R3);)
15  _END_KERNEL("shift_right");
```

Listing 2.3: OPINCAA Kernel Example

The `OPINCAA` library was designed in order to simplify the work of the programmers, but they still need to have deep knowledge and understanding og the `ConnexArray` architecture.

OPINCAA - ConnexArray Simulator

Besides of the human friendly assembly language provided by `OPINCAA` library, it also provides a `ConnexArray` simulator that can be used for development and debugging software in the absence of the accelerator. What it does is to create a set of pipe files that usually exposed by the `ConnexArray` accelerator. The programmers can use the simulator interchangeable with the real devices, without any modification.

In our project, we used the `OPINCAA` simulator in the development and validating phases.

Chapter 3

Compiler Infrastructure

Writing computer vision applications for ConnexArray meant writing everything from scratch in binary code. This approach was limitative and involved a tremendous effort and deep understanding of how the architecture works. Furthermore, typical open source applications couldn't be run on this architecture without manually translating the code to ConnexArray assembly.

When we designed the system, we had in mind the following characteristics:

- modularity
- easy to extend
- no hard-constrains for a programming language

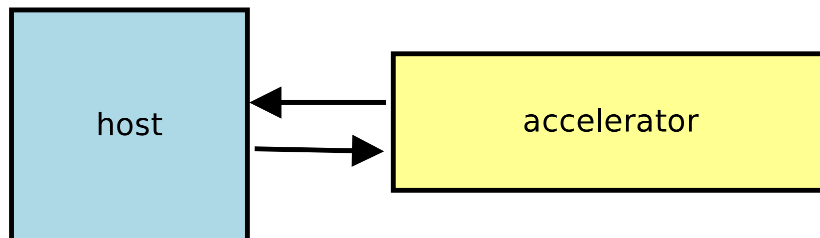


Figure 3.1: Host-Accelerator System

`OpenCL` uses a `C API` on the host machine and a separate high-level kernel language for code definition. The kernel is the code that will run on the heterogeneous computers, accelerators (figure 3.1). The language used in the kernels has explicit support for vectorization so it is highly recommended for `SIMD` architectures. By using `OpenCL`, the programmer must write separate code for the host and accelerator, which can be handled by different compilers.

Figure 3.2 shows the compilation process and the execution flow. Because the host and the accelerator have different architectures, the compilers used may be different.

The `ISA` provided by `ConnexArray` doesn't have control instructions and most off the values are directly hard-coded inside the instructions. Arguments, like size of the vector, can be passed only through the Local Store memory and can produce a memory

waste, since the data must be duplicated among each cell's Local Store memory. In order to avoid such a waste, we choose to create a runtime compilation process for the accelerator. The avoiding mechanism is extensively described in chapter 4.

Our project aims to create a compiler for the `ConnexArray` architecture.

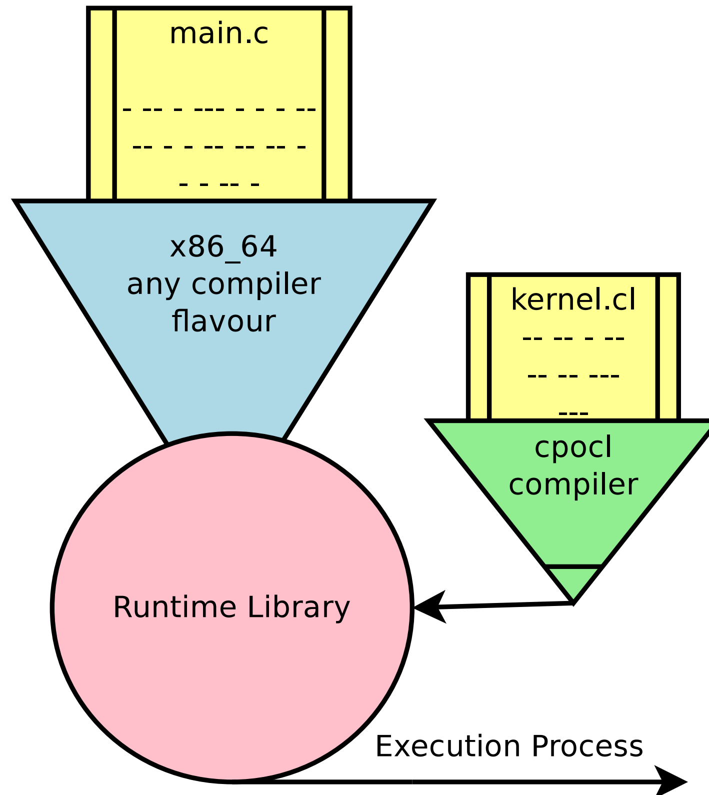


Figure 3.2: Compilation Process and Execution Flow

3.1 Host Compiler

`OpenCL` provides a standard mechanism in which applications can be run. The host in responsible for offloading computation and instrumenting communication with the accelerator. The compilation can be done with any compiler as far as the system has access to an `OpenCL` library. The `OpenCL` library provides the standard `API` accessible from the host and has implemented accelerator particularities, so called *runtime library* (details in chapter 4).

```
processor       : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
gcc-version     : 4.8.1
clang-version   : 3.4
```

Listing 3.1: Host Machine Processor and Compilers Available

In our testing scenarios we used a x86_64 processor, as shown in listing 3.1. The host compiler doesn't need to be aware of the accelerator existence, even less of its architecture particularities.

3.2 Accelerator Compiler

In order to provide support for `ConnexArray`, the compiler must handle two mandatory components:

- the runtime library - enhanced `OpenCL` library with `ConnexArray` support; this provides capabilities of orchestrating the communication between host and accelerator: memory mapping, data transfers, instruction fetching, etc.; this component is exhaustively described in chapter 4
- code generator - the actual compiler that generates machine dependent code; this component is exhaustively described in chapter 5

Another component that we will include in the project is taking advantage of the frequency scaling at instruction level of the `VASILE` architecture. The approach and details information are presented in chapter 6.

Chapter 4

Runtime Library

The compiler infrastructure runtime library presented in figure 3.2 implements OpenCL standard API and makes the connection between the host and the accelerator. Each hardware manufacturer implements the standard through an SDK that is usually closed source (at least for the moment). The standard is open and architecture independent. It uses a C API on the host machine and a separate high-level kernel language for code definition. The kernel is the code that will run on the heterogeneous computers.

We have chosen to use POCL[8], an open source implementation of the OpenCL standard which can easily be adapted for new devices by implementing or using already existing backend from the LLVM infrastructure.

4.1 Implementation

We extended the POCL library with support for a new device, called *"connex"*. In order to do so, the device had to define some generic hooks in the struct pocl_device_ops structure, as shown in listing 4.1.

```
void pocl_connex_init_device_ops (struct pocl_device_ops *ops) {
        ops->device_name = "connex";

        ops->init_device_infos = pocl_connex_init_device_infos;
        ops->uninit = pocl_connex_uninit;
        ops->init = pocl_connex_init;
        ops->malloc = pocl_connex_malloc;
        ops->free = pocl_connex_free;
        ops->read = pocl_connex_read;
        ops->write = pocl_connex_write;
        ops->run = pocl_connex_run;
}
```

Listing 4.1: struct pocl_device_ops Initialization

Each hook is a architecture specific implementation of different operations of the OpenCL API and has the following meaning:

- *init_device_infos* - configures OpenCL library with different parameters that are architecture specific like: the existence of a compiler, the number of processing units, memory types, memory sizes, etc.
- *init* - initialize the accelerator; for ConnexArray, this hook is responsible of opening the 4 communication pipes of the accelerator
- *uninit* - uninitialize the accelerator: close pipes, free any additional memory allocated on the host machine
- *malloc* - create buffers on the accelerator; this hook is representative for the memory mapping model and will be discussed in section 4.2
- *free* - frees buffers; will also be discussed in section 4.2
- *read* - initiate data transfer from the accelerator to the host; this hook will be discussed in section 4.3
- *write* - initiate data transfer from the host machine to the accelerator; this hook will be discussed in section 4.3
- *run* - this hook is responsible of the runtime compilation that will be discuss in chapter 5 and also for fetching instructions to the accelerator, like we will see in section 4.4

The accelerator architecture is chosen by using the POCL_DEVICES environment variable. The programmer doesn't have to be aware of these hooks when writing code. **Appendix B.1** is presenting a simple OpenCL program that increments the values of a given array, puts the result to another memory location and, at the end, performs a sanity check verification on the host machine. As you can see, the application is using the OpenCL API. In table 4.1 you can see the correspondence between the OpenCL API and the POCL hooks.

OpenCL API call	**POCL hook**
clCreateCommandQueue	pocl_connex_init
clReleaseCommandQueue	pocl_connex_uninit
clCreateBuffer	pocl_connex_malloc
clReleaseMemObject	pocl_connex_free
clEnqueueReadBuffer	pocl_connex_read
clEnqueueWriteBuffer	pocl_connex_write
clEnqueueNDRangeKernel	pocl_connex_run

Table 4.1: Correspondence between OpenCL API and POCL hooks

4.2 Memory Mapping Model

Figure 5.1 shows the generic memory model of the OpenCL memory mapping. The accelerator can make use of 4 types of memory: the global memory, the constant memory which is often incorporated in the global memory, the local memory which is available through the execution units of the same workgroup and private memory which is available only to the execution unit itself.

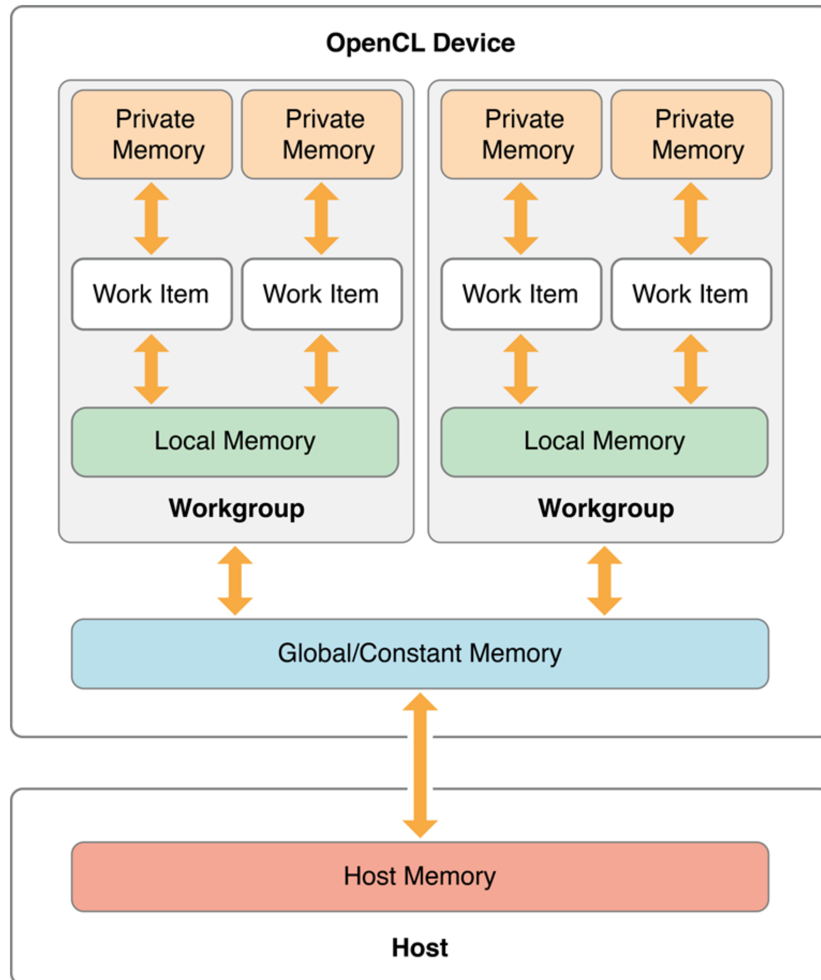


Figure 4.1: OpenCL Generic Memory Model [1]

ConnexArray accelerator has a different memory model, so the mapping between the OpenCL generic model and the one implemented in our work in not straight through. Each processing unit has a Local Store of 1024 cells. The memory is designed as a matrix: the columns are represented by the processing units, while the lines are represented by the available cells. A processing unit doesn't have access to the entire Local Store; it can access only the elements placed on its column.

If the application is sending a vector with 128 elements, each processing unit will receive

[1]image from https://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide (15.08.2014, 23:42)

in its accessible Local Store one value. If the vector is larger, the wrapping is done transparently to the user by allocating more lines in the Local Store. The compiler is aware of the wrapping and will know to generate more code in order to process other elements as well.

Global and Constant Memory

Since `ConnexArray` doesn't have a shared memory among the execution units, we didn't do a 1-to-1 mapping of the memory. To simulate the global memory behavior, we copied the data allocated in the global-constant memory area among each Local Store so that each processing unit can have access to those values. If the data is not constant, writing and syncing data among the processing units can be a very expensive operation so we decided to not have support for global memory if the values aren't constant.

Local Memory

The `ConnexArray` architecture is not suitable for pairing execution units in work-groups and sharing data between them. In our model, each workgroup consist of a single execution unit. We mapped this zone with the Local Store.

Private Memory

The private memory is a memory zone accessible only by a processing unit. We mapped this zone with the register files, so that the library can expose direct access to the registers for experimented users.

4.3 Data Transfers

The `ConnexArray` Local Store can be written and read through the I/O interface. There are two pipes in the I/O interface: an inbound `FIFO` used to transfer data from the host to the accelerator and an outbound `FIFO` used to transfer data from the accelerator to the host.

Any transfer is initiated by pushing the I/O descriptor into the inbound `FIFO`. The I/O descriptor contains 3 fields: a writing bit, an address field and a count field. If the writing bit is set, then the operation will be a write one, otherwise it will be a reading operation. The address field specifies the Local Store address from where the transfer must start and the count field specifies the number of vector to transfer minus 1.

The sequences between the accelerator and the host memory are transferred through `DMA` in parallel with the computational process.

Because the operations are async I/O, if the transfer is `write` the host must check the transfer completion by reading a single word from the outbound `FIFO`.

The I/O operations are initiated by calling *clEnqueueReadBuffer* or *clEnqueueWrite-Buffer*. These functions receive accelerator memory addresses which are abstracted by the `cl_mem` data type.

4.4 Instruction Fetching

`ConnexArray` has a separate pipe for receiving instructions from the host machine. Although the instructions are executed immediately when they are received by the accelerator, the `ConnexArray` also has a buffer where it memories the last 1024 instructions received. This is done because the architecture has one control instruction (`ijmpnzdec` - see Appendix A.1) that allows the accelerator to jump back in the instruction stream.

Because the instructions have a fix size (as shown in listings 2.1 and 2.2), it is not necessary to include a header or anything else additional when the instructions are fetch through the pipe.

4.5 Validation Results

The validation of the entire runtime library was done using the simulator provided by the `OPINCAA` library. We wanted to prove the accuracy of the results of two features: the data transfer and the instruction fetching process.

4.5.1 Accuracy of Data Transfers

To prove the correctness of data transfers, we used a slightly modified version of the code listed in Appendix B.1. Since the compiler was not finished at this moment, we weren't able to actually execute code. So we create a program that writes the data to the accelerator and reads back the data to another memory location. The results were identical.

We also wanted to validate the data transfer with more complex tests that also execute instructions on the accelerator. So, we modifies some of the testing scenarios of the `OPINCAA` library as follows:

Testing Write Operation

We made our program to deal with the write operations and removed those operations from the `OPINCAA` library. After running our program, we run the test through `OPINCAA` simulator. The results were always positive.

Testing Read Operation

We made our program to deal with the read operations and removed those operations from the `OPINCAA` library. After running the test through the `OPINCAA` simulator, we

run our program to get back the values after the computation. The results were always positive.

4.5.2 Accuracy of Instruction Fetching

To prove the correctness of instruction fetching, we created several tests. Each test consisted of a collection of instructions written directly in ConnexArray's machine code. Since at this moment the code generator was not finished, we by-passed that functionality and made to framework to read our manually created tests and fetch the instructions through the pipe to the OPINCAA simulator.

We also changed the OPINCAA simulator and make it to dump the instructions received in the human readable format provided by OPINCAA library.

The output dumped by the OPINCAA simulator was identical with the expected one and so we validated the instruction fetching process.

Chapter 5

Backend: Code Generator

One of the reason we chose to use POCL is that it relies on a suite of tools provided by LLVM framework.

The runtime library has support for generating LLVM IR from the kernel's source code. The LLVM IR is a high-level language, very similar with assembly languages, but agnostic in respect of any architecture flavour. It is a strongly typed language and abstracts the calling convention through *call* and *ret* instructions and explicit arguments. Another significant difference is that LLVM IR uses an infinite set of registers that have to be mapped to real registers in the code generation phase.

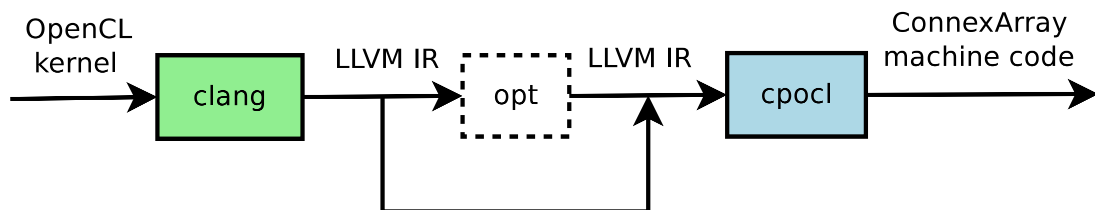


Figure 5.1: Code Generation Diagram Flow

The code generation is formed by two mandatory stages and one optional stage, as presented in figure 5.1. The first stage is to generate LLVM IR from the kernel's source code. The last stage is to generate ConnexArray machine code from LLVM IR. These two stages are mandatory. In between, there can be several number of other stages that perform optimizations. The optimizations are done exclusively on the LLVM IR and can be architecture dependent.

5.1 Generation of LLVM IR

The first stage of code generation is to generate LLVM IR from the kernel's source code. This stage is architecture independent and it's implemented directly in POCL's library core.

Listing 5.1 shows a simple OpenCL kernel example that receives two memory addresses

and stores at the second address the first value incremented.

```
1  __kernel void kernel(short *in, short *out) {
2          short i = get_global_id(0);
3          out[i] = in[i] + 1;
4  }
```

Listing 5.1: OpenCL Kernel: Increment Values

Listing 5.2 shows the LLVM IR representation of the same kernel, as it was generate by clang compiler through POCL library. This representation will be taken as input for other stages.

```
1  define void @kernel(i16 addrspace(3)* %in, i16 addrspace(1)* %out) {
2  entry:
3    %idx = tail call i16 @_Z13get_global_idj(i16 0)
4    %arrayidx = getelementptr i16 addrspace(3)* %in, i16 %idx
5    %0 = load i16 addrspace(3)* %arrayidx
6    %add = add i16 %0, 1
7    %arrayidx4 = getelementptr i16 addrspace(1)* %out, i16 %idx
8    store i16 %add, i16 addrspace(1)* %arrayidx4
9    ret void
10 }
```

Listing 5.2: LLVM IR Kernel: Increment Values

5.2 Generation of ConnexArray Machine Code

The last stage of code generation is to generate ConnexArray machine code from LLVM IR. This stage is the backend-core of the compiler (cpocl) and it's done only after every optimization was applied.

First step, before starting to generate target-dependent code, was to transform the LLVM IR code into Static Single Assignment (SSA) [10] form. This format simplifies and improves the results of different compiler techniques by simplifying the properties of variables. The SSA form ensures that each variable is assigned only once, and every variable is defined before it is used. If a variable is used among different basic blocks, the variable is split into different versions. The different versions among basic blocks can be gathered together into new variables by using the so called PHI nodes.

5.2.1 Code Generation Design

LLVM framework provides a special tool for generating target-independent code. The tool can be extended with architectural particularities in order to create a compiler

backend. However, we were not able to use this tool since we need to pass through different values at run-time to our compiler backend.

What we used was the `LLVM` Pass Framework, designed to perform optimizations, analysis and transformations. There are several types of passes that one can implement, depending on the purpose in which the tool will be used: *ModulePass*, *FunctionPass*, *LoopPass*, *BasicBlockPass*, etc. We used the *FunctionPass* since the kernel representation in `LLVM IR` had only a function. The pass has to extend the *FunctionPass* class and implement the virtual function *runOnFunction*. The pass must be register and will be available inside the **`opt`** tool as any other pass implementation.

Figure 5.2 shows the class diagram of out `LLVM` pass. *ConnexCodeGen* is the main class of the pass and it's responsible of instrumenting the interaction between other classes.

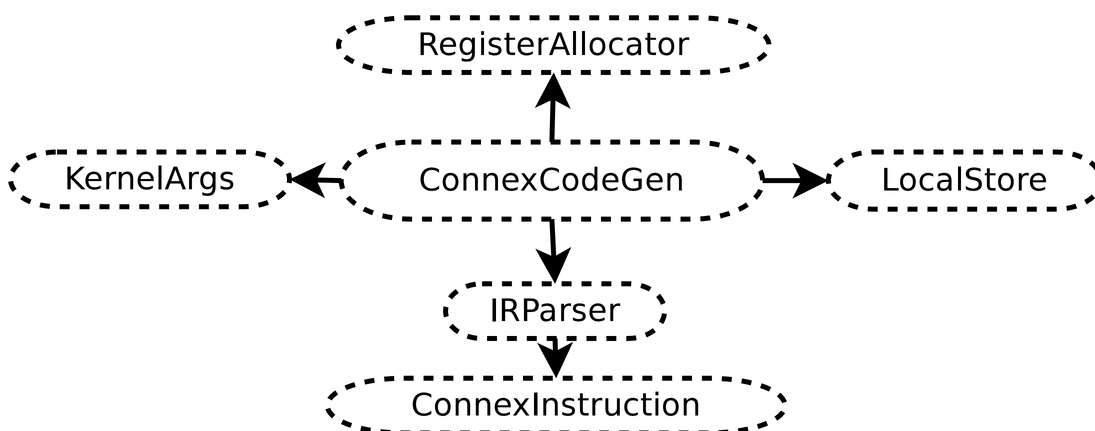


Figure 5.2: Backend Classes Diagram

KernelArgs class is responsible of receiving data from the runtime library and match the kernel arguments with Local Store addresses or store their fixed values into registers, as applicable.

The *LocalStore* class is responsible of instrumenting `getelementptr` instructions from `LLVM IR`. This class has to catch different semantics of computing addresses. Table 5.1 shows the different semantics of this instruction. As we can see in the last line of the table, this instruction might require to perform additional shift instructions in order to read data from a neighbor cell.

getelementptr	**Local Store**
getelementptr %addr, 0	LS[0]
getelementptr %addr, 128	LS[1]
getelementptr %addr, 1	LS[0] from the right cell

Table 5.1: `getelementptr` Semantics

IRParser is the class responsible of translating `LLVM IR` code into `ConnexArray` ma-

chine code. The ConnexArray instructions are modeled inside the *ConnexInstruction* class. By default, the machine code will be dumped in a binary file named *"kernel.cnx"*. The filename can be change at compile time by using the option *-output="filename"*.

Since the ConnexArray has a limited number of registers and LLVM IR models an infinite register machine, an implementation of register allocation was mandatory. The algorithm is further discussed in section 5.2.2.

5.2.2 Register Allocator

Because the ConnexArray machine has only 32 registers for each processing element, we needed to implement a register allocator algorithm that will assign a large number of LLVM IR registers to the registers available in ConnexArray. To do the assignment, the compiler must determine which variables are live at the same time. The common approach is to construct a graph such that every vertex represents a unique variable in the program and reduce the problem to a *K-coloring* algorithm[3], where K is the number of registers available. Variables can live at the same moment only if they receive different colors.

The graph coloring algorithm is a **NP-complete** [7] problem and so is the register allocation. Even if those algorithms can produce efficient code, their execution time is high. The compilation process in our case is done at runtime so the fast register allocation is a significant aspect. In recent years, a new allocation technique was proposed by Wimmer et al. [21]. They proposed a single pass over the list of variable live ranges and assigning to registers only the variables with a short lifetime. The other variables are spilled into memory. However, each execution unit of the ConnexArray has access to only 1024 memory locations and this approach may not have the best results.

Hack et al. [5] published a paper in which they make a formal demonstration of the fact that the register allocation problem can be resolved in polynomial time for programs in SSA format. They showed that interference graphs of SSA programs are chordal and such they can pe colored in polynomial time. We choose to implement Hack's algorithm version presented in his thesis [6].

5.3 Evaluation and Results

The scenario we used is a custom bubble-sort algorithm optimized to take advantage of the ConnexArray architecture. We choose this algorithm because even if it may look like a classical algorithm, it's not one suitable for the architecture. The processing elements of ConnexArray have access only to their Local Store and it's a costing operation to reach out elements stored in other processing elements Local Stores. The algorithm will make use of a significant number of register in order to measure the penalty introduce by the compiler.

5.3.1 Validation

We verified the correctness of the compilation process by running different sized vectors with random elements through the bubble-sort algorithm.

The algorithm use a sequence of two kernels: *kernel_ odd* compares elements stored on odd positions, while *kernel_ even* compares elements stored on even positions (listing 5.3. The reduce call will gather information about how many interchanges were done during the kernels execution. The kernels are applied until no exchange are done, namely until the reduce value is zero.

```
1  __kernel void kernel_odd(short *in) {
2          short i = get_global_id(0);
3          short exchange = 0;
4
5          if (i % 2) {
6                  short a = in[i - 1];
7                  short b = in[i];
8                  if (b < a) {
9                          exchange = 1;
10                         in[i] = a;
11                 }
12                 if (exchange)
13                         in[i - 1] = b;
14         }
15         reduce(exchange);
16 }
17
18 __kernel void kernel_even(short *in) {
19         short i = get_global_id(0);
20         short exchange = 0;
21
22         if (i % 2 == 0 & i != 0) {
23                 short a = in[i - 1];
24                 short b = in[i];
25                 if (b < a) {
26                         exchange = 1;
27                         in[i] = a;
28                 }
29                 if (exchange)
30                         in[i - 1] = b;
31         }
32         reduce(exchange);
33 }
```

Listing 5.3: OpenCL Kernel: Modified Bubble Sort Algorithm

5.3.2 Evaluation

We measured different times in order to compute the compiler performance. To extract relevant information, we ran a test through the `OPINCAA` simulator with 128,000 elements that occupies almost the entire Local Store memory. We also put the elements in reverse order so that the computational time would be as worst as possible. The times obtained are listed in table 5.2 and are the averages obtained from 5 different runs.

	Time
Test 0	0.021 s
Test A	0.973 s
Test B	1.224 s
Test C	1.311 s

Table 5.2: Bubble-Sort Execution Time on a Vector with 128,000 Elements

Test 0 represents the execution time of a serial implementation on the host machine.

Test A represents the execution time measured inside the `OPINCAA` simulator. The difference between this test and *Test 0* is huge, but we should keep in mind that the scenario is not appropriate for the `ConnexArray` architecture since it needs to access and modify addresses from others execution units Local Stores.

Test B represents the execution time of the algorithm implemented inside the `OPINCAA` library. We can see that there is a overhead introduced by the library. This overhead exists because the library must translate the assembly language into `ConnexArray` machine code. The overhead introduced by `OPINCAA` library is of 25.79%.

Test C represents the execution time of the `OpenCL` implementation. The overhead introduced by our compiler is of 34.73%, with only 7.10% worse than *Test B*. The penalty is small, but the programmer work has been significantly reduced.

Chapter 6

Frequency Scaling at Instruction Level

VASILE (**V**ector **A**rchitecture for **S**caling at **I**nstruction **LE**vel) [14] is a derived architecture from ConnexArray implemented in FPGA technology that enables the adjustment of the frequency at instructions per second in order to enhance performance.

VASILE includes a **frequency selection unit (FSU)** to ensure different delays through the IUs. The FSU is formed by several clock sources, a clock MUX and the associated logic. Each clock source corresponds to one or more instruction classes. The instruction are inspected during the decode phase and then, the FSU selects the required clock source. The Instruction Fetch Decode and Dispatch Unit (IFDDU) ensures that the decoded instructions do not reach the PEs before the clock MUX switches to the required clock source for the instruction. This is done by adding a **configurable delay line**. The switch time is device-dependent and may require a relatively lengthly time to complete.

In order to gain maximum performance for the VASILE architecture, a specialised compiler is required. The application must be profiled in advance to identify the most common instructions. The accelerator is than calibrated with different frequencies in order to gain the maximum performance. In our approach, the compiler reads the calibration from an input file in order to perform instruction rearrangement.

6.1 Testing scenario

For the performance evaluation of the compiler that takes into consideration the frequency scaling at instruction level capabilities, we chose the Sum of Squared Differences (SSD) algorithm because it is a widely used algorithm in computer vision applications.

Listing 6.1 shows the OpenCL's kernel source code. Each processing element has access to 1024 memory locations, so we arranged the data transfers in such way that each processing element will compute the SSD algorithm for two vectors of 500 elements length.

```
__kernel void kernel(short *a, int n, short *b, int m) {
        short i, j;

        short id = get_global_id(0);
        for (i = 0; i < n; i++)
                for (j = 0; j < m; j++) {
                        short x = a[id + i * 128];
                        short y = b[id + j * 128];

                        short z = x - y;
                        z = z * z;

                        reduce(z);
                }
}
```

Listing 6.1: OpenCL Kernel: SSD Algorithm

6.2 Hardware Calibration Phase

At this stage, we were able to run the implementation from listing 6.1 and profile the application. For testing we used a mid-range Zynq 7020[1]. As mentioned in section 2.2, the distances between multiplier columns on the target device are higher so the expected performance for this type of instructions is lower. The baseline frequency is 125 Mhz. Table 6.1 shows the maximum frequencies obtained when running the `SSD` algorithm on`VASILE` and the percentage of the instruction grouped by their type. In this case, the switching penalty will be at most:

$$T_{switch} <= 3 * max(T_1, T_2)$$

where T_1 corresponds to F_1 (117 Mhz) and T_2 corresponds to F_2 (160 Mhz).

Instructions	**Max. Frequency (Mhz)**	**How many (%)**
Addition/Subtraction	160	54%
Multiplication/Division	117	34%
Memory Access	160	12%

Table 6.1: SSD Profiling Results

[1] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices/index.htm

6.3 Loop Tiling Optimization

In order to increase the performance and take advantage of the hardware capabilities exposed by VASILE, the instructions need to be clustered in order to minimize the number of switches between instruction classes.

As we can see, the SSD algorithm doesn't have data dependencies among different loop iterations. In our implementation, the compiler performs an Instruction Level Parallelism (ILP) analysis in order to identify the loops that respect this property.

Each loop forms a different basic block in LLVM IR representation so the unused registers can be used in order to perform loop tiling. The compiler will try to use all of the free registers in order to create bigger clusters of instructions at the same type. Listing 6.2 shows the pseudo-code of how the compiler is applying this optimization.

```
1  for i = 1, n do
2          for j = 1, m/30 do
3                  R[0] = a[i]
4                  for k = 0, 29 do
5                          R[1 + k] = b[30 * j + k]
6                  end for
7                  for k = 0, 29 do
8                          R[1 + k] = R[1 + k] - R[0]
9                  end for
10                 for k = 0, 29 do
11                         R[1 + k] = R[1 + k] * R[1 + k]
12                 end for
13                 for k = 0, 29 do
14                         reduce R[1 + k]
15                 end for
16 end for
```

Listing 6.2: Tiled SSD Algorithm

Before applying the optimization, the compiler must determine if the optimization will gain any speedup or not. The optimization is done only if the speedup is significant. However, even if the speedup is not significant, the computation of the profitability of the algorithm is still done and will add additional time to the execution.

6.4 Evaluation

We measured different times in order to compute the compiler performance. The times obtained are listed in table 6.2 and are the averages obtained from 5 different runs. We obtain a **speedup of 1.178**.

	Time
Test 0	0.141 s
Test A	1.397 s
Test B	1.186 s

Table 6.2: SSD Execution Time

Test 0 represents the execution time of a serial implementation on the host machine.

Test A represents the execution time of the algorithm on `ConnexArray` architecture.

Test B represents the execution time of the algorithm on `VASILE` architecture.

Chapter 7

Conclusion and Future Development

In this thesis, we presented the techniques behind the process of creating a compiler for `ConnexArray` architecture that is modular, easy to extend and without hard-constrains for a programming language. It also presents the importance of the fact that the compiler must be aware of the hardware capabilities of the architecture and explore them, as we saw happening in the case of `VASILE` architecture.

We were able to build the compiler from scratch for `ConnexArray` architecture, prove its functionality and measure the penalty time introduced by the compiler.

We were also able to create awareness of the `VASILE` architecture capabilities and obtain a speedup of 1.178.

The compilers research fields are manifold and many possible avenues for research remain. We identified a list of subjects that can be further investigated:

- changing the front-end of the compiler and measuring the performance
- minimize the execution time of the code generation
- improving the Frequency Scaling algorithm
- improving the Register Allocation algorithm
- exploring other techniques of gaining performance for computer vision applications

Appendix A

Instruction Set Architecture (ISA) for ConnexArray

Mnemonic	Description	Opcode
nop	No operation	000000000
red	Launch reduction with R[left]	100000000
iwrite	LS[Immediate Value] = R[left]	110010
iread	R[dest] = LS[Immediate Value]	110100
write	LS[R[right]] = R[left]	100010100
read	R[dest] = LS[R[right]]	100100100
vload	R[dest] = Immediate Value	110101
ldix	R[dest] = INDEX	100100000
endwhere	Enables all cells	100011111
wherecry	Load Carry Flag into Cell Enable	100011100
whereeq	Load Equal Flag into Cell Enable	100011101
wherelt	Load Less Flag into Cell Enable	100011110
mult	Initiate R[left] * R[right]	100001000
multlo	R[dest] = low half of multiplication results	100101000
multhi	R[dest] = high half of multiplication results	100111000

Mnemonic	Description	Opcode
cellshr	Shift Register = R[left] then shift right by R[right] positions	100010001
cellshl	Shift Register = R[left] then shift left by R[right] positions	100010010
ldsh	R[dest] = Shift Register	100110000
add	R[dest] = R[left] + R[right]	101000100
sub	R[dest] = R[left] - R[right]	101010100
addc	R[dest] = R[left] + R[right] + Carry	101100100
consdub	R[dest] = (R[left] - R[right) ? 0 : R[left] - R[right]	101110100
eq	R[dest] = (R[left] == R[right]) ? 1 : 0	101001000
ult	R[dest] = (R[left] < R[right]) ? 1 : 0 (unsigned)	101101000
lt	R[dest] = (R[left] - R[right]) ? 1 : 0	101011000
shl	R[dest] = R[left] « R[right]	101000000
ishl	R[dest] = R[left] « right	101000001
shr	R[dest] = R[left] » R[right]	101010000
ishr	R[dest] = R[left] » right	101010001
shra	R[dest] = R[left] »> R[right]	101100000
ishra	R[dest] = R[left] »> right	101100001
not	R[dest] = R[left]	101001100
or	R[dest] = R[left] | R[right]	101011100
and	R[dest] = R[left] & R[right]	101101100
xor	R[dest] = R[left] R̂[right]	101111100
setlc	LC = Immediate Value	010101

ijmpnzdec	Require:
	Immediate Value < 1023
	if (LC != 0):
	PC = PC - Immmediate Value
	LC = LC - 1
	if (LC == 0):
	PC = PC + 1
	LC reverts to initial value
		010011

Table A.1: Instruction Set Architecture for ConnexArray

Appendix B

Simple Example of OpenCL Program

```
1  #define DATA_SIZE (2048)
2
3  const char *KernelSource = "                                   \n" \
4      "__kernel void square(                                     \n" \
5      "   __constant short* input,                               \n" \
6      "   __global short* output)                                \n" \
7      "{                                                         \n" \
8      "   short i = get_global_id(0);                            \n" \
9      "   output[i] = input[i] + 1;                              \n" \
10     "}                                                         \n";
11
12 int main(int argc, char **argv)
13 {
14         cl_int err;
15
16         short data[DATA_SIZE];
17         short results[DATA_SIZE];
18         unsigned int correct;
19
20         size_t global;
21         size_t local;
22
23         cl_uint num_platforms;
24         cl_platform_id platform;
25         cl_device_id device_id;
26         cl_context context;
27         cl_command_queue commands;
28         cl_program program;
29         cl_kernel kernel;
30
```

```
31         cl_mem input;
32         cl_mem output;
33 
34         int i = 0;
35 
36         /* Initialize data. */
37         for (i = 0; i < DATA_SIZE; i++) {
38                 data[i] = rand() % 100;
39                 if (data[i] < 0)
40                         data[i] = -data[i];
41         }
42 
43 
44         /* Get number of supported platforms. */
45         err = clGetPlatformIDs(0, NULL, &num_platforms);
46 
47         /* Get platforms. */
48         cl_platform_id *platforms;
49         platforms = (cl_platform_id *)calloc(num_platforms, sizeof(
               cl_platform_id));
50         err = clGetPlatformIDs(num_platforms, platforms, NULL);
51         if (err != CL_SUCCESS) {
52                 printf("Error: Failed to get platform IDs!\n");
53                 return EXIT_FAILURE;
54         }
55         platform = platforms[0];
56 
57         /* Connect to CPU. Use CL_DEVICE_TYPE_GPU for GPU. */
58         err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &
               device_id, NULL);
59         if (err != CL_SUCCESS)  {
60                 printf("Error: Failed to create a device group!\n");
61                 return EXIT_FAILURE;
62         }
63 
64         /* Create context. */
65         context = clCreateContext(0, 1, &device_id, NULL, NULL, &err)
               ;
66         if (!context) {
67                 printf("Error: Failed to create a compute context!\n"
                       );
68                 return EXIT_FAILURE;
69         }
70 
71         /* Create a command queue. */
72         commands = clCreateCommandQueue(context, device_id, 0, &err);
73         if (!commands) {
```

```
74                  printf("Error: Failed to create a command queue!\n");
75                  return EXIT_FAILURE;
76          }
77
78          /*  Create the compute program from the source buffer. */
79          program = clCreateProgramWithSource(context, 1, (const char
                **)&KernelSource, NULL, &err);
80          if (!program) {
81                  printf("Error: Failed to create compute program!\n");
82                  return EXIT_FAILURE;
83          }
84
85          /* Build the program executable. */
86          err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
87          if (err != CL_SUCCESS) {
88                  size_t len;
89                  short buffer[2048];
90
91                  clGetProgramBuildInfo(program, device_id,
                        CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &
                        len);
92                  return EXIT_FAILURE;
93          }
94
95          /* Create the compute kernel in the program we wish to run.
                */
96          kernel = clCreateKernel(program, "square", &err);
97          if (!kernel || err != CL_SUCCESS) {
98                  printf("Error: Failed to create compute kernel!\n");
99                  return EXIT_FAILURE;
100         }
101
102         /* Create the input and output arrays in device memory for
                our calculation. */
103         input = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(
                short) * DATA_SIZE, NULL, NULL);
104         output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(
                short) * DATA_SIZE, NULL, NULL);
105         if (!input || !output) {
106                 printf("Error: Failed to allocate device memory!\n");
107                 return EXIT_FAILURE;
108         }
109
110         /* Write our data set into the input array in device memory.
                */
111         err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0,
                sizeof(short) * DATA_SIZE, data, 0, NULL, NULL);
```

```
112         if (err != CL_SUCCESS) {
113                 printf("Error: Failed to write to source array!\n");
114                 return EXIT_FAILURE;
115         }
116
117         /* Set the arguments to our compute kernel. */
118         unsigned short count = DATA_SIZE;
119         err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
120         err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
121         if (err != CL_SUCCESS) {
122                 printf("Error: Failed to set kernel arguments! %d\n",
                        err);
123                 return EXIT_FAILURE;
124         }
125
126         /* Get the maximum work group size for executing the kernel
                on the device. */
127         err = clGetKernelWorkGroupInfo(kernel, device_id,
                CL_KERNEL_WORK_GROUP_SIZE, sizeof(local), &local, NULL);
128         if (err != CL_SUCCESS) {
129                 printf("Error: Failed to retrieve kernel work group
                       info! %d\n", err);
130                 return EXIT_FAILURE;
131         }
132
133         /* Execute the kernel over the entire range of our 1d input
                data set
134            using the maximum number of work group items for this
                  device. */
135         global = count;
136         err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &
                global, &local, 0, NULL, NULL);
137         if (err) {
138                 printf("Error: Failed to execute kernel!\n");
139                 return EXIT_FAILURE;
140         }
141
142         /* Wait for the command commands to get serviced before
                reading back results. */
143         clFinish(commands);
144
145         /* Read back the results from the device to verify the output
                . */
146         memset(results, 0, count * sizeof(short));
147         err = clEnqueueReadBuffer(commands, output, CL_TRUE, 0,
                sizeof(short) * count, results, 0, NULL, NULL );
148         if (err != CL_SUCCESS) {
```

```
149                 printf("Error: Failed to read output array! %d\n",
                        err);
150                 return EXIT_FAILURE;
151         }
152 
153         /* Validate our results.*/
154         correct = 0;
155         for(i = 0; i < count; i++) {
156                 if(results[i] == data[i] + 1)
157                         correct++;
158         }
159 
160         /* Print a brief summary detailing the results. */
161         printf("Computed '%d/%d' correct values!\n", correct, count);
162 
163         /* Shutdown and cleanup. */
164         clReleaseMemObject(input);
165         clReleaseMemObject(output);
166         clReleaseProgram(program);
167         clReleaseKernel(kernel);
168         clReleaseCommandQueue(commands);
169         clReleaseContext(context);
170 
171         return 0;
172 }
```

Listing B.1: Simple Example of OpenCL Program

Bibliography

[1] Munshi Aaftab. The OpenCL Specification, version 1.0. *The Khronos Group*, 2008.

[2] Călin Bîră, Lucian Petrică, and Radu Hobincu. OPINCAA: A light-weight and flexible programming environment for parallel SIMD accelerators. *Science and Technology*, 16(4):336–350, 2013.

[3] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[4] James Fung and Steve Mann. Using graphics devices in reverse: GPU-based image processing and computer vision. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 9–12. IEEE, 2008.

[5] Sebastian Hack and Gerhard Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.

[6] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in ssa-form. In *Compiler Construction*, pages 247–262. Springer, 2006.

[7] Ian Holyer. The np-completeness of edge-coloring. *SIAM Journal on Computing*, 10(4):718–720, 1981.

[8] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, pages 1–34, 2014.

[9] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[10] Peter Lee, Frank Pfenning, and André Platzer. Static single assignment.

[11] Rainer Leupers. *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools.* Springer Publishing Company, Incorporated, 2010.

[12] Ivan Llopard, Albert Cohen, Christian Fabre, Jérôme Martin, Henri-Pierre Charles, and Christian Bernard. Code generation for an application-specific VLIW processor with clustered, addressable register files. In *Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems*, pages 11–19. ACM, 2013.

[13] Mihaela Maliţa, Gheorghe Ştefan, and Dominique Thiébaut. Not multi-, but many-core: designing integral parallel architectures for embedded computation. *ACM SIGARCH Computer Architecture News*, 35(5):32–38, 2007.

[14] Lucian Petrica, Valeriu Codreanu, and Sorin Cotofana. Vasile: A reconfigurable vector architecture for instruction level frequency scaling. In *Faible Tension Faible Consommation (FTFC), 2013 IEEE*, pages 1–4. IEEE, 2013.

[15] Nalini K Ratha and Anil K. Jain. Computer vision algorithms on reconfigurable logic arrays. *Parallel and Distributed Systems, IEEE Transactions on*, 10(1):29–43, 1999.

[16] Angel Rodríguez-Vázquez, Fernando Medeiro, and Edmond Janssens. *CMOS Telecom Data Converters*. Springer, 2003.

[17] Gheorghe Stefan. The ca1024: A massively parallel processor for cost-effective hdtv. In *Spring Processor Forum: Power-Efficient Design*, pages 15–17, 2006.

[18] Gheorghe Stefan, Anand Sheel, Bogdan Mitu, Tom Thomson, and Dan Tomescu. The ca1024: A fully programable system-on-chip for cost-effective hdtv media processing. In *Hot Chips: A Symposium on High Performance Chips*, 2006.

[19] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[20] Marco Alexander Treiber. *Optimization for Computer Vision*. Springer, 2013.

[21] Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179. ACM, 2010.

[22] Yuanrui Zhang, Lanping Deng, Praveen Yedlapalli, Sai Prashanth Muralidhara, Hui Zhao, Mahmut Kandemir, Chaitali Chakrabarti, Nikos Pitsianis, and Xiaobai Sun. A special-purpose compiler for look-up table and code generation for function evaluation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1130–1135. European Design and Automation Association, 2010.