ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and
Electrical Engineering

Spring Semester 2020

LaTeX Report Template

Semester Project / Master Project

Pierre-Hugues BLELLY
pblelly@student.ethz.ch

May 2020

Supervisors: Matheus Cavalcante, matheusd@iis.ee.ethz.ch
Samuel Riedel, sriedel@student.ethz.ch

Professor: Prof. Luca Benini, lbenini@ethz.ch

Acknowledgements

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor. For a detailed version of the declaration of originality, please refer to Appendix **??**

Pierre-Hugues BLELLY,
Zurich, May 2020

Contents

1 Introduction **1**
- 1.1 Heterogeneous systems . . . 1
- 1.2 Design Issue with heterogeneous systems . . . 2
- 1.3 Currently Available Workflow for Halide . . . 2

2 Preliminaries / Background **4**
- 2.1 Hero . . . 4
- 2.2 Halide Language . . . 4
  - 2.2.1 Programing model . . . 4
  - 2.2.2 Debugging Options . . . 5
  - 2.2.3 Basic Scheduling Options . . . 5
  - 2.2.4 Porting Halide to new Platforms . . . 6
- 2.3 Compilation Workflow . . . 6
- 2.4 The full hero platform . . . 7

3 Design Implementation **8**
- 3.1 Schedule Implementation . . . 8

4 Results **10**
- 4.1 Test Setup . . . 10
- 4.2 Comparaison between OpenMp and Halide on the different platforms . . . 10

5 Conclusion and Future Work **11**
- 5.1 First Section . . . 11
- 5.2 Second Section . . . 11

List of Figures

List of Tables

Chapter 1

Introduction

Thanks to system integration and the better energy efficiency of modern Central Processing Unit (CPU), embedded systems are becoming quite powerful. They can now manage multiple sensors in real-time. But the power consumption is still an issue. These chips often have a limited power budget as their goal is to be embedded in small systems where the battery capacity doesn't exceed the hundreds of milliAmpere hours.

To improve the energy efficiency and the computing power of those chips, researchers have been trying to find new architecture that would suit those applications. Heterogeneous computing may be one of the solutions to this problem. Heterogeneous systems are often composed of one general-purpose CPU and multiple coprocessors which will be used to handle specific tasks [**?**]. This strategy has been used in the SoC industry by ARM since 2011 by ARM in it's big.LITTLE architecture [**?**]. This architecture based on two clusters of ARM Cortex A7 and A15, was designed to increase the computing power in embedded systems such as smartphones while increasing the battery life of the device. Even though this architecture relied on a single Instruction Set Architecture (ISA), the idea was to use more powerful cores when needed and turn them off when the device was in sleep mode or when the lower power core could handle the task.

These architectures are interesting to increase the power efficiency but can also increase the processing power at the same time. Researchers have been trying to take advantage of multiple ISA and leverage their specificities to increase the overall performances with great success. In this article [**?**] from 2014, researchers compared homogeneous and heterogeneous systems based on three ISA, under strict design constraint (such as area or thermal dissipation) they observed that the heterogeneous designs performed better than their homogeneous counterpart. Heterogeneous systems are also heavily used in Data Centers, as the processing needs keep rising, they now use GPU as they are heavily optimized for parallel tasks and are now powerful enough to be used in scientific applications.

Hero [?] is a heterogeneous system developed by the Integrated Systems Laboratory (IIS) and the Energy Efficient Embedded Systems (EEES). This platform is composed of a hard multicore ARM 64 Juno System on Chip (SoC) (composed of two Cortex A57 and four Cortex A553) and up to eight Parallel Ultra Low Power (PULP) clusters (composed of eight RI5CY cores), running on an Field Programmable Gate Array (FPGA).

1.1 Design Issue with heterogeneous systems

Due to their heterogeneous nature, those systems are difficult to design and program. During their conception, numerous design choices need to be made to make sure that all the CPU in the system will interact will each other, these choices will impact the peak performance of the design or it's power consumption [?]. The architect has to choose how the different accelerators will interact, how they will share data, maybe extend the existing ISA to distribute tasks, and so on.

The software design isn't easy either, when compiling for such systems, the compiler needs to create an executable that will run on the host processor, but also integrate instructions that will be run on the coprocessors. Most of the time, the programmer assign which part of the code will run on which accelerator, OpenMP uses this model via the `#pragma` preprocessor instructions. Then during the compilation, llvm will compile the code bits to the right platform and link them using the adequate linker.

1.2 Currently Available Workflow for Halide

Currently Hero supports OpenMP, which is an Application Programming Interface (API)which "defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer" [?]. This API has been implemented on hero to easily take advantage of the clusters. The toolchain uses clang, and the clang-offload-bundler to compile the applications. Clang uses a custom front head for hero which supports all the available configurations for hero (only the pulp cluster for simulation, with the ARM CPU as the host for the FPGA or with a riscV 64 bits CPU).

Exploring the design space using OpenMp's directive isn't perfect, and require the developer to adapt its code to run with a specific schedule, for example, to take advantage of vectorization, the programmer has to manually unroll the loops and change it's code so that the compiler knows what part of the code will be using the Single Instruction Multiple Data (SIMD) instructions. This approach leads to an important development time but also an extensive testing process whenever the schedule is changed to ensure that the resulting code works as intended. This approach makes testing for a new schedule pretty difficult and inefficient.

Starting from this idea that separating the algorithm from how it is run, researchers from the Massachusett Institute of Technology (MIT), created Halide. Halide [**?**] is a programming language that was designed to allow the developer to explore multiple design choices quickly by separating the algorithm from the execution schedule. This language was designed to be used in image or array processing applications. Every processing pipeline designed with Halide has two parts. The first part consist of the functional description of the processing kernel, this is the algorithm that will be executed on the array. The second part is the schedule of the pipeline. This schedule describes how the algorithm will be executed on the system. This programming model is interesting because the developer can implement the algorithm without having to take into account the boundaries of the functions or the border effects. Then he can quickly bound the different variables of the pipeline and design it's schedule afterward. All the constraints will be asserted during the compilation of the pipeline without any intervention from the developer. The scheduling process can even be done automatically during the compilation by the library, in order to find an optimal schedule on the target platform.

Chapter 2

Preliminaries / Background

2.1 Hero

2.2 Halide Language

2.2.1 Programing model

Halide is a functionnal programming language, embedded into C++ designed to write high performance image and array processing code [**?**]. This language uses a functionnal paradigm to describe the functionnalities of the processing pipeline. The code of the algorithm is separated from how it will be implemented on the target (schedule).

Every pipeline is a function (`Halide::Func` composed of other functions and expressions (`Halide:expr`). These two objects use special variables (`Halide:Vars`) to describe the operation executed on the array. The code snippet 2.1 describe a basic pipeline which compute the distance of each coordinate of a two-dimentionnal array from on position specified by the vector `(center_x, center_y)`.

```
Halide::Var x, y;
Halide::Param center_x, center_y;
Halide::Expr offset = Halide::pow(x - center_x, 2)
                    + Halide::pow(y - center_y, 2);

  gradient(x, y) = offset;
```

Listing 2.1: Simple Pipeline Example

This simple pipeline only has one stage, but it is possible to create multiple stage pipeline and transform it into a single stage inlined pipeline or keep it's multi stage structure, this operation will be done during the scheduling phase.

After designing the pipeline, we can define it's schedule via the different directive included in Halide. Halide implements all the basic scheduling option like parallelizing, unrolling the loops, splitting one loop into an inner and an outer loop... These options will be described in the section Basic Scheduling Options.

In the example 2.2, we can see how the scheduling works. All instructions are function of the pipeline object, and the final pipeline will implement these instructions. The example shows a simple schedule applied on our gradient, this schedule consists of parallelizing the execution over the x axis, and unrolling along the y axis.

```
gradient.parallel(x);
gradient.unroll(y, 10);
```

Listing 2.2: Simple Pipeline Example

To execute the pipeline, Halide provides a large range of options, we can execute it directly using the `.realize(x_max, y_max)` function, this is useful for debugging purpose, but most of the compile options are targeted at cross-compilation.

As the initial goal of Halide was to target devices such as CPU, the library is capable of compiling the pipeline to a lot of different platform and output format. Halide support translation to C code, llvm assembly file, or already compiled object file specific to a given target(Cuda, Arm, Risc-V, MIPS, PowerPc...), and a given operating system(Linux, Mac, Windows, Android). The developer can also chose to compile the pipeline to a library to use in another application.

2.2.2 Debugging Options

Halide has tools to debug the pipeline during it's compilation or when it is executed. First of all, the `print()` and `print_when()` functions can be called at any time in a pipeline and allow to print values of some variables. Another useful tool is the `.trace_store()` function which prints the value every functions evaluated in the pipeline. It is possible to get more informations during the compilation of the pipeline by setting the `HL_DEBUG_CODEGEN` to 1, this will output the stages of the compilation and a pseudo code representation of the pipeline. Finally, variables and functions can have a label, which will be used by halide in it's internal representation, this function greatly reduce the debugging time of the schedules as Halide gives every variables a different name from the source code.

2.2.3 Basic Scheduling Options

Halide implement different scheduling instruction, and some of them just reshape the code in a different way. These scheduling instructions are useful to prepare the code for other instrutions (such as parallelization or vectorization), but also to take advantage of memory locality.

Non Architecture Specific Instrutions

- Reorder : Tells halide how to traverse the domain of the pipeline stage (ffor example in a column major or row major way)
- Split : Split a dimension along inner and outer subdimensions.
- Tile : Cut the domain in tiles.
- Fuse : Join two dimensions in a single fused dimension.
- Unroll : Unroll along one dimension.

These instructions are the basic one, but it is also possible to split the code in arbitrary areas which can have arbitrary shapes and sizes using reduction domains and the `where()` instruction.

Some of the primitives such as vectorize or parallel need to be implemented on the target platform as they take advantage of the specificities of the system. The halide Team doesn't prive a clear guide on how to port the language to a new platform, but using the header generated when building the pipeline, we can get enough information to implement enough functions to port halide to this new platform.

2.2.4 Porting Halide to new Platforms

First of all, to compile to a specific platform, we need a build of llvm which support the desired architecture. Then we can look in the pipeline header file to list all the vital functions for our pipeline. We can also use the error message when linking the pipeline to determine which functions we need to implement on the target platform. Currently only the memory allocation functions, the print functions and the task distribution functions are implemented, and they are enough to test basic pipelines such as matrix multiplications or light image modifications. After the implementation we can work on the compilation workflow for hero.

2.3 Compilation Workflow

Every application has at least two source files, one C++ file which will generate the object file of the pipeline, the main application. Currently, we can only compile the application to the hardware simulator. The compilation has two phases, during the first one, we compile the Halide application using llvm and run it on the host platform, this application will then generate an risc-V object file and a header. Then we compile the hero application using the already available Makefile, we include the header in the main application and the object file to the sources during the linking command.

2.4 The full hero platform

The hardware platform has a more complex compiling process, currently the code is distributed to the PULP thanks to OpenMp. The compilation first generate the llvm representation of the code, then assign space on the device via `hc-omp-space`, and also `clang-offload-bundler` to distribute generate the llvm assembly code for the right platform. Finally the program uses clang to compile the application, thanks to the special hero target, clang links every function correctly and then embed the riscV code inside the ARM application.

Chapter 3

Design Implementation

To test halide on hero, I used two benchmark. The first one was a basic gradient example, and the second one a matrix multiplication pipeline that I took in the provided examples and then adapted to be used in a hero application. The matrix axample is more interesting, because it represent what a typical signal processing application may do. It is also quite easy to benchmark with different sizes to see the impact of the memory access on the execution time.

```
ImageParam A(type_of<int>(), 2);
ImageParam B(type_of<int>(), 2);
Var x, y;
Func matrix_mul("matrix_mul");
Func out;

RDom k( 0,A.width() );

matrix_mul(x, y) += A(x, k) * B(k, y);

out(x, y) = matrix_mul(x, y);
```

Listing 3.1: Matrix Multiplication Pipeline

3.1 Schedule Implementation

Most of the schedules implemented on halide doesn't require any platform specific implementation as they are only unrolling, splitting or swapping loops. During my project I used two platform specific schedules: vectorization and parallelization. The `.vectorize(x)` instrucion unrolls one loop in assembly, and the vectorization is done by g++ using the SIMD instructions of the chip. For hero, the simd extention wasn't supported by g++

but, we could still use this instructions as it reduced the number of jumps and thus the total execution time.

The `.parallel(x)` instructions uses two functions: `halide_do_par_for` and `halide_do_par_for_fork`. `halide_do_par_for` adds the tasks to the task queue of the pulp cluster, every task will execute `halide_do_par_for_fork` on the corresponding core (if the core id is equal to the task number modulo the number of available cores). Every task consist of a part of the processing pipeline.

Chapter 4

Results

4.1 Test Setup

I benchmarked two applications on two platforms. I benchmarked the halide port on the hardware simulation for the PULP cluste, and one openMp matrix multiplication application on the developpement platform on a Xilinx ZCU102. For both application, I generated random matrices of différent sizes, and for each sizes multiplication I counted the number of cycles needed to complete the operation. With this setup, we can easily compare the performances of halide and OpenMp in a real world scenario for at least two basic schedules: Single threaded and Multi Threaded. To give the results a more meaning I also calculated the number of operations per cycles where one operation can either be an addition a multiplication or a memory access (which take 2 cycles each), so for a matrix of size n, the number of operations to finish the multiplication is : $6 * n * *3 + n * *2$ (each coefficient needs $2n$ memory accesses, $2n$ additions and multiplications and one memory store).

4.2 Comparaison between OpenMp and Halide on the different platforms

Chapter 5

Conclusion and Future Work

Draw your conclusions from the results you achieved and summarize your contributions. Comparisons (e.g., of hardware figures) with related work are also appropriate here. Point out things that could or need to be investigated further.

5.1 First Section

5.2 Second Section