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

Implementation of a Heterogeneous System for Image Processing on an FPGA

Semester Project

Pierre-Hugues BLELLY
pblelly@student.ethz.ch

May 2020

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

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

Acknowledgements

Abstract

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 B

Pierre-Hugues BLELLY,
Zurich, May 2020

Contents

List of Figures

List of Tables

List of Acronyms

AARCH64 . . . 64 bit ARM architecture

API . . . . . Application Programming Interface

CI . . . . . Continuous Integration

CPU . . . . . Central Processing Unit

CUDA . . . . . Compute Unified Device Architecture

EEES . . . . . Energy Efficient Embedded Systems

ETH Zürich . . . Eidgenössische Technische Hochschule Zürich

FPGA . . . . . Field Programmable Gate Array

GPU . . . . . Graphic Processing Unit

HERO . . . . . Heterogeneous Embedded Research Platform

IIS . . . . . Integrated Systems Laboratory

ISA . . . . . Instruction Set Architecture

LLVM . . . . . Low Level Virtual Machine

MIPS . . . . . Microprocessor without Interlocked Pipelined Stages

MIT . . . . . Massachusetts Institute of Technology

OpenCL . . . . . Open Computing Language

List of Acronyms

OpenMP . . . . Open Multi-Processing

PMCA . . . . . Programmable Many Core Accelerator

PULP . . . . . Parallel Ultra Low Power

RTL . . . . . Register Transfer Level

SIMD . . . . . Single Instruction Multiple Data

SoC . . . . . System-on-Chip

ULP . . . . . Ultra Low Power

Chapter 1

Introduction

Thanks to the smaller nodes of modern lithography technologies and the transistor density we can achieve with them, modern low-power Central Processing Units (CPUs) can have a large amount of cores while keeping their power consumption under a few Watts. A single Raspberry Pi 3 has a peak performance of 6 GFLOP/s for a power consumption of only 7 W [1]. Embedded systems can take advantage of this increase in efficiency to become more autonomous and not rely on an external computer for heavy computation. We can find this type of architecture on some nano-drones such as the CrazyFlie 2.0 [2], which can be extended with additional shields. Using a custom shield, the Integrated Systems Laboratory (IIS) of ETH Zürich managed to analyze a video signal in real-time and train a neural network for autonomous navigation [3]. The computing unit achieved a rate of 562 MFLOPS/s on a power-enveloppe of only 45 mW. These results were achieved thanks to the heterogeneous architecture of the drone. To keep its power consumption low, the dronee wakes-up the Ultra Low Power (ULP) chip of the shield only during computation. The shield use a Parallel Ultra Low Power (PULP) cluster, which is a RISC-V based System-on-Chip (SoC) which can be configured with up to eight cores, this chip provides the computing power needed to analyze the video. With this configuration, the energy consumption of the CrazyFlie stays low, the autonomy when using the shield only drops by ten seconds compared to when the shield is turned off [3].

In more general terms, heterogeneous systems are composed of multiple coprocessors all managed by a host processor. This architecture is interesting when it comes to embedded systems, as it is possible to achive greater energy efficiency than homogeneous systems. If each coprocessor has been designed to solve a certain task, it can achieve energy efficiency than a general purpose CPU. According to Venkar and Tullsen [4], under heavy design constraints (such as die area or therman dissipation), systems using multiple Instruction Set Architectures (ISAs) achieved better performances than their best homogeneous counterpart.

This strategy has been used in the SoC industry by ARM since 2011 [5]. The big.LITTLE architecture is based on two clusters of ARM Cortex A7 (the "LITTLE" cores) and A15 (the "big" cores), and was designed to increase the computing power in low-power systems such as smartphones while increasing the battery life of the device. This architecture relied on a single ISA (ARMv7). The goal was to use the more powerful cores during heavy computation or graphic rendering, and let the low-power cores handle the background tasks or manage the device during sleep. Currently, every smartphone SoC manufacturer use the big.LITTLE architecture or a similar technology.

Even in data centers, where power consumption is also an issue, Graphic Processing Units (GPUs) are used thanks to their massive core count and the various Application Programming Interfaces (APIs) such as Compute Unified Device Architecture (CUDA) or Open Computing Language (OpenCL) which simplify the developement process for GPU accelerators.

Heterogeneous Embedded Research Platform (HERO) [6] is a heterogeneous system developed by the IIS of ETH Zürich and the Energy Efficient Embedded Systems (EEES) of the University of Bologna. This platform is composed of a hard macro ARM 64 CPU and up to eight PULP clusters running on an Xilinx ZYNC ZC706 Field Programmable Gate Array (FPGA).

This platform is designed to "facilitate rapid exploration on all software and hardware layers" [6] and includes a heterogeneous compilation toolchain with support for Open Multi-Processing (OpenMP), an API developed to make developement of multi threaded applications easier [7]. This API implements new preprocessor instructions to tell the compiler how to execute the code on the system.

1.1. Design Issue with heterogeneous systems

During their conception, numerous design choices need to be made specifically how the CPUs in the system will interact will each other. These choices will impact the peak performance of the design and its power consumption [4]. The computer architect has to choose how the different Programmable Many Core Accelerators (PMCAs) will interact, how they will share data and maybe extend the existing ISAs to distribute tasks. The software design is challenging, when compiling for heterogeneous platforms. The compiler needs to create an executable that will run on the host processor, but also dedicate parts of the final binary to embed the code that will be distributed on the PMCAs.

Even though APIs such as CUDA, OpenMP or OpenMP did a great job at making the overall developement easier, most of the work is still done by hand. The programmer has to handle memory mapping: depending on the system architecture, the data may be stored in the shared memory or on the PMCA extra space , and the tasks needs to be scheduled by hand, and distributed on the correct PMCA. Moreover, the code is often not portable as some schedule are target-dependant. An algorithm coded with CUDA

will only run on a GPU, so the code cannot be reused for another platform. Porting APIs to new platform is not trivial, and might require months of work to port it to a new target.

1.2. Currently Available Workflow for HERO

Currently, HERO supports OpenMP [8], an 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" [9]. This API has been implemented on HERO to easily take advantage of the PULP clusters. The toolchain uses the Clang compiler [10] to compile the applications. HERO uses custom Clang front-ends to support all the available configurations (only the PULP cluster for simulation with the ARM host CPU or with a 64 bits RISC-V CPU).

To distribute the code, OpenMP uses preprocessor instructions to tell Clang where the code will run and how it will be executed. Exploring the design space using OpenMP's directives can be time-consuming. For example, the developer must explicitly tell which part of the code to offload. Trying to change the order of multiple loops may cause bugs in the algorithm, and complex schedules often impact code readability making them harder to debug.

Halide [11] was proposed to explore the idea of separating the algorithm from how the code will be executed on the target (the schedule). This separation makes testing different schedules easier on the developer, as the algorithm code will stay the same, and only the scheduling will be changed when testing. Every processing pipeline designed with Halide has two parts. The first part consists of the functional description of the processing kernel, i.e., the algorithm that will be executed. The second part is the schedule of the pipeline. The programer will explicitely tell Halide how the pipeline should be executed. Thanks to specific function calls, the developer can decide whether the code will be run on multiple threads or a single one, change the order of execution of different parts, split or unroll them. The developer also has the freedom to implement any schedule without having to change the main algorithm. This programming model is interesting because the developer can quickly implement the algorithm without having to take into account the boundaries of the inputs, and then work on an optimal schedule, or quickly adapt it if the algorithm needs to be executed on another platform.

The intermediate variables can be bounded afterwards if needed, and the main variables such as characteristics of the inputs are automatically bounded by Halide. An image processing pipeline will only compute the output on the pixels of the input. 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.

The goal of this project was to port Halide to HERO, and execute image processing kernels on the HERO system running on an FPGA. First Halide needs to be compiled

to support RISC-V and compile basic applications to the hardware simulation. From then we can work on the heterogeneous compilation to support the current HERO test platform.

This report is organized as follow, beside this introductory section, in Section 2 we discuss about the HERO platform and the Halide language. In Section 3, we explain how we managed to port Halide to the simulation platform, and the compilation workflow we designed to create Halide applications. Then in Section 4, we benchmark our implementation, and compares it's performance against an already implemented API. The last Section concludes the report and outline the next steps to improve the implementation.

Chapter 2

Background

Overview of the content, to be redacted

2.1. HERO

The HERO platform is an heterogeneous platform available in different configurations. This platform is composed of a hard macro multicore ARM 64 Juno SoC (composed of two Cortex A57 and four Cortex A53 cores) and up to eight PULP clusters (each of them using up to eight RI5CY cores [6]), running on an FPGA (a Xilinx ZYNC ZC706). PULP is a cluster of CPU based on the RISC-V ISA, an open source ISA designed to support a wide range of platform from embedded systems to accelerators in datacenters [12]. The modularity of the ISA makes it an interesting for PMCAs as the core are designed to support only the useful instruction for the task we want to run, which make them small and energy efficient. The system is also available with a Ariane RISC-V 64-bits core, or just as an independent PULP cluster for hardware simulation. The system uses 256 KiB of L1 scratchpad data cache, coupled with 256 KiB of L2 data and instruction cache and a 4 KiB of L1 cache.

HERO has a fully functional software toolchain with "support for OpenMP, a linux driver and runtime libraries for both the host and the PMCA" [6]. The toolchain uses `clang`, a C compiler front-end of Low Level Virtual Machine (LLVM). The heterogeneous compiling is done by separately compiling the part of the code and then bundling together inside a single binary the two compiled file during the linking phase of the host [8]. The final binary uses the host ISA, and embed the PMCA code in dedicated sections of the binary.

2.2. Halide Language

2.2.1. Programing model

Halide is a functional programming language embedded into C++, designed to write high performance image and array-processing code [13]. This language uses a functional paradigm to describe the processing pipeline, and dissociate the array-processing code from its schedule (how the code will be compiled and run on the system).

Every pipeline is a function (`Halide::Func`) built using other functions and expressions (`Halide:expr`) or variables (`Halide::Vars`). The code Listing 2.1 describe a basic pipeline which computes the distance of each coordinate of a two-dimensional array from a given position `(center_x, center_y)`. The creation of the pipeline is straightforward, we only need to write the desired operation using the variables `x` and `y`. During the execution of the pipeline or it's compilation, Halide will bound `x` and `y` according to the size of the output.

```
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 multi-stage pipelines and schedule them as desired. They can be transformed into a single-stage inlined pipeline or kept as is. The different stages can be scheduled to start as soon as they have enough data, or wait for the previous one to finish before starting to compute.

Scheduling is done via basic scheduling primitives implemented by Halide. The primitives consist of basic code transformations such as loop unrolling or reordering, loop splitting or merging variables together into a single one. More advanced instructions like parallelization or vectorization are also available. These instructions can be combined as needed to create complex schedules. Section 2.2.3 explains the most important scheduling instructions in more details.

The Listing 2.2 shows how schedule are designed. All instructions are a function of the pipeline object, they can be executed on any variable of the pipeline. Some instructions need the variables to be bounded (e.g., the vectorize instruction) before using them. The scheduling primitives can be combined as needed, and the programmer can also create intermediate variables via those primitives to control precisely the execution of the code.

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

Listing 2.2: Simple Schedule Example.

In the Listing 2.2, Halide creates one task per value of `x`. These tasks will be executed in parallel on all the cores of the system. Every task will execute a single loop over the `y` axis, but instead of computing only one value of the output of the pipeline per iteration, the task will compute ten values per iteration.

The pipeline can be translated or compiled by Halide to be executed directly on the compilation computer or in another application. The pipeline can be immediatly executed using the function `.realize(x_max, y_max)`. If an output buffer of the correct size is provided, Halide will execute the pipeline over the rectangular domain `(0,0),(x_max,y_max)`. As Halide was designed primarily to work with different hardware platforms, the cross-compilation process has been simplified, and the pipeline can be translated to other languages. Halide support translation to C code, LLVM assembly file, or already compiled object file specific to a given target (CUDA, ARM, RISC-V, Microprocessor without Interlocked Pipelined Stages (MIPS), PowerPc...), and a given Operating Systems (Linux, Mac, Windows, Android). The pipeline can also be exported as a static library to use in another application.

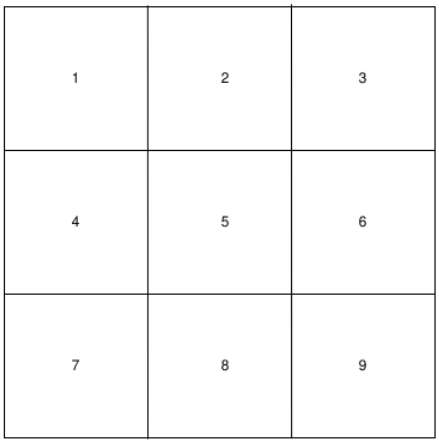
2.2.2. Debugging Options

Halide provides tools to debug the pipelines, and debugging tips to help the developers [14]. The `print` instructions prints the value of a variable at any point of the pipeline, `print_when()` only print when a boolean condition is True. The `.trace_store()` function keeps a trace of every function evaluation during execution, as long as the function has not been inlined, the parameters and the result of the function call will be stored in the trace and printed after the exeuction.

Halide can print more information on the screen during the compilation of the source code by setting the environemental variable `HL_DEBUG_CODEGEN` to 1. Halide will output information about every stages of the compilation and a pseudo code representation of the pipeline loops. Finally, variables and functions can be labeled. Halide will replace the generic name of the variable with the label when printing the pseudo code or when using `gdb`.

2.2.3. Basic Scheduling Options

Every Halide schedule applies a simple modification to the source code. Every instruction affects one or multiple variables. There are no limitation to the complexity of the schedule or the number of variable inside a pipeline.



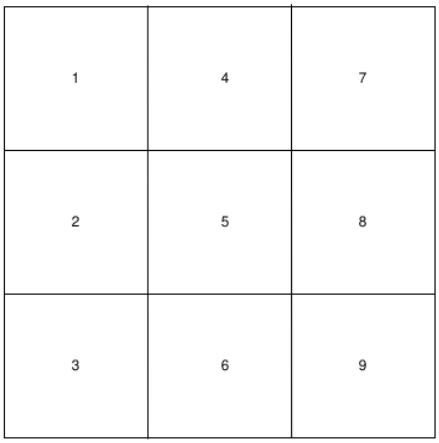

```
# Generated loops
for(int y = 0; y < 3; y++){
  for(int x = 0; x < 3; x++){
    do something on (x,y)
  }
}
```

Figure 2.1.: Base Schedule.

Default Schedule

If no schedule is specified, Halide will evaluate the pipeline in the same order as it's arguments. The first variable being the inner most loop, and the last one the outer most loop. In Figure 2.1, Halide will compute the output of the pipeline in a row major fashion.

Reorder




```
  pipeline.reorder(y,x);
```

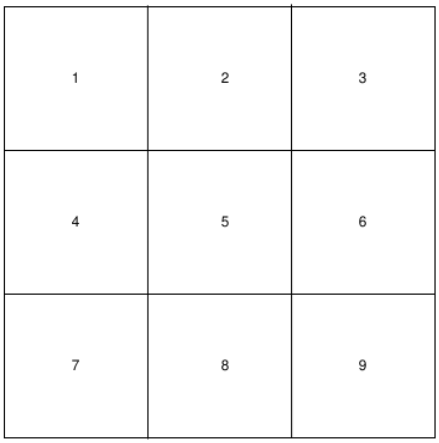
```
# Generated loops
for(int x = 0; x < 3; x++){
  for(int y = 0; y < 3; y++){
    do something on
    (x,y)
  }
}
```

Figure 2.2.: Reorder Schedule.

The `.reorder` instruction reorders the variable to have the given nesting order, starting from the innermost. In the Figure 2.2, the array is now processed in a column major

fashion.

Fuse

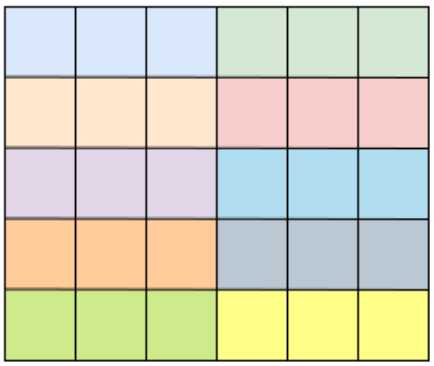



```
  pipeline.fuse(x,y,xy);


# Generated loops
for(int xy = 0; xy < 9; xy++){
  do something on
  (xy)
}
```

Figure 2.3.: Fused Schedule.

The `.fused` instruction fuses two dimensions together, transforming a two-dimensionnal array into a one-dimensionnal array.

Split



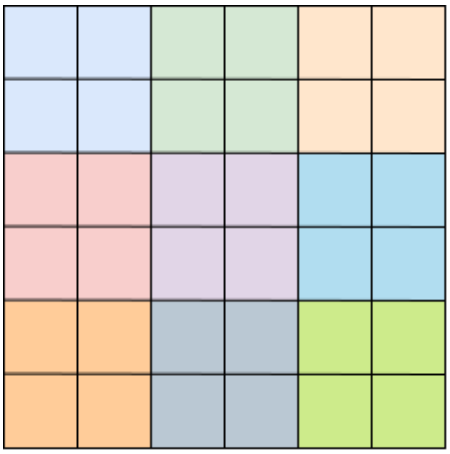
```
pipeline.split(x,x_o,x_i, 3);


# Generated loops
for(int y = 0; y < 6; y++){
  for(int x_o = 0; x_o < 2; x_o++){
    for(int x_i = 0; x_i < 3; x_i++){
      do something on
      (x_o * 3 + x_i, y)
    }
  }
}
```

Figure 2.4.: Split Schedule.

This schedule split a loop in an inner and an outer subdimensions, where the size of the inner dimension is specified by the last argument. This shedule is useful to cut the array

in smaller pieces that will be computed in parallel or using Single Instruction Multiple Data (SIMD) instructions.

Tile

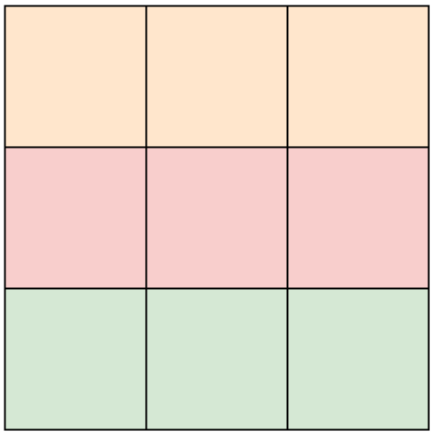


```
pipeline.split(x,y,x_o,y_o,x_i,y_i,2,2);


# Generated loops
for(int y_o = 0; y_o < 6; y_o++){
  for(int x_o = 0; x_o < 3; x_o++){
    for(int y_i = 0; y_i < 2; y_i++){
      for(int x_i = 0; x_i < 2; x_i++){
        do something on
        (x_o * 2 + x_i, y_o * 2 + y_i)
      }
    }
  }
}
```

Figure 2.5.: Tile Schedule.

The Tile schedule is similar to the Split schedule, but along two dimensions. It creates multiples smaller rectangular tiles which can be processed independently.

Unroll

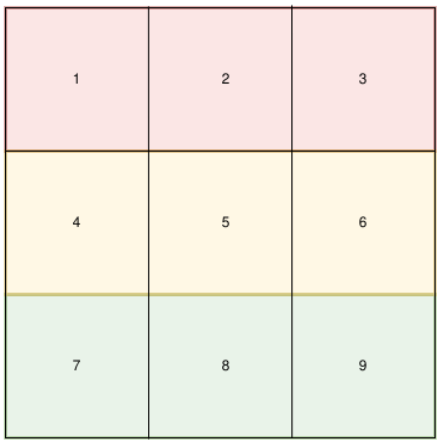


```
# Generated loops
pipeline.unroll(x,3);
for(int y = 0; y < 4; y++){
  do something on (0,y)
  do something on (1,y)
  do something on (2,y)
}
```

Figure 2.6.: Unroll Schedule.

The Unroll schedule unrolls the code along one dimension. This technique is often used when multiple computations share the same data, to prevent multiple memory access. In the Figure 2.6, we first split the x dimension before unrolling as Halide cannot unroll a variable if it is not bounded.

Parallel




```
pipeline.parallel(y);
# Core 0: y = 0 for(int x = 0; x < 4; x++)
    {
  do something on (x,0)
}

# Core 1: y = 1
for(int x = 0; x < 4; x++){
  do something on (x,1)
}

# Core 2: y = 2
for(int x = 0; x < 4; x++){
  do something on (x,2)
}
}
```

Figure 2.7.: Parallel Schedule.

The parallel schedule distributes the pipeline to all the available cores.

In the Figure 2.7, the code is distributed on three cores, each of them execute a single loop along the y axis.

Halide will create a task for each value the variable can take, and these tasks will be executed with the `halide_do_par_for` function. This function needs to be overwritten on HERO to distribute the tasks on the PULP cluster.

Vectorize

The goal of this schedule is to setup the code so to make use of the SIMD instructions of the CPU. Currently, LLVM does not support the SIMD extension implemented in the PULP cluster, but the generated code will take advantages of all the registers available to compute the output values, and try to compute multiple values at the same time.

Chapter 3

Design Implementation

3.1. Porting Halide to new targets

Halide programs relies on the LLVM libraries to compile code for the desired targets. To build the Halide library, we first need to build LLVM with the correct flags and add the support for the building machine. As the HERO toolchain already has a build of this compiler, we can use it to compile Halide, but the `-DBUILD_SHARED_LIBS` flag has to be disabled, as Halide does not support shared libraries. We added a `make` target to main `Makefile` of the project, to simplify the installation process.

Once the installation process was complete, we worked on porting Halide to HERO on the hardware simulation. This was the first step of the project, as this platform is easier to work with than the full HERO platform. We do not have to work with the heterogeneous toolchain, so the compilation process is simpler. Unlike the hardware platform, the hardware simulator only simulate a single PULP cluster configured with eight cores. The first core of the cluster acts as the host, it first initializes the cluster and then starts the execution of the application. When a fork is executed on the other cores, the host core continues the execution like the other cores.

3.2. Compilation Workflow

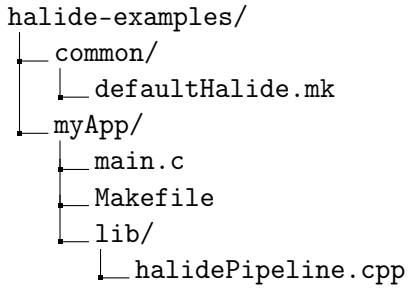


Figure 3.1.: Directory structure for Halide applications.

Every application follows the directory structure described in Figure 3.1, the `common` folder is shared between all the applications and contains a common `Makefile` that will be included in every application's `Makefile`. The source code is split between two files, the pipeline generator in the `lib/` folder, and the main HERO application. The pipeline generator uses Halide to generate the pipeline object file which will be linked during the compilation of the application.

Figure 3.2 shows the whole compilation process to build a Halide application for HERO. The compilation is done in two steps. First, we compile the Halide program to execute on the build computer. This program is then run on the machine and creates a RISC-V object file and a header file. This header file must be included in the HERO application (`main.c`). The application building process is based on the OpenMP workflow, we used the same `Makefile` with some modification to link the pipeline with the main application.

We first compile the source code to LLVM assembly code. Then due to the heterogeneous nature of the system, a custom program: `hc-omp-pass`, changes every part of the code that may cause issue due to architectural differences between the host and the PMCA (on HEROv3, the host can be a 64-bit RISC-V or an ARM processor, and the pointers have to be changed due to the incompatibilities between the 32-bit and 64-bit addressing). In our case, as we only compile for the PULP cluster, this step does not affect the code, but it will be useful to support the full HERO platform. Then, we use the LLVM assembly files coupled with the pipeline object file to generate the final binary.

The header file generated by Halide declares every function available on Halide and required to have a fully working Halide implementation. Most of them do not use platform-specific functionalities, and will be compiled to RISC-V without any issue. But others such as memory management functions or thread distribution functions, which require dedicated functions in the PULP runtime have to be overwritten to work on the cluster. Finally, functions that are only called when using the Just in Time compilation or other advanced functionalities, have to be overwritten, but we can implement simple

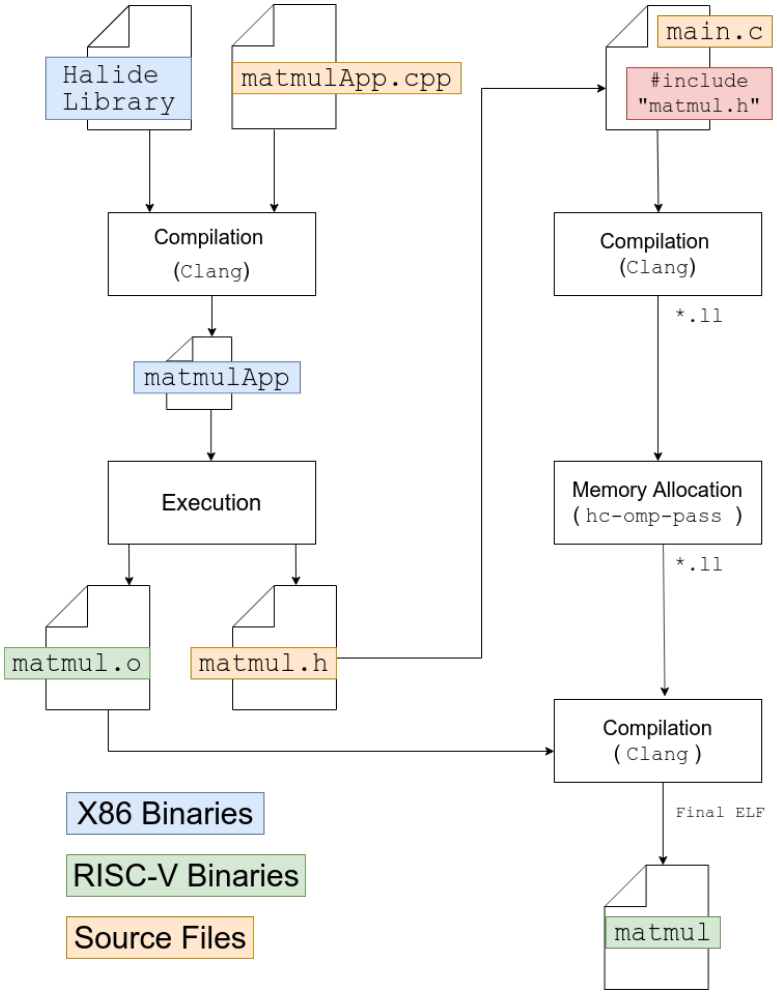


Figure 3.2.: Compilation Workflow for an Halide Application.

stub functions to prevent linking errors, as we do not need those functionalities for our pipelines. The comments in the header precisely describes the role of each function and under which circumstances they have to be overwritten. We implemented the necessary functions in the PULP runtime to make the parallel schedule work, as this schedule is essential to take advantage of the high core count of the cluster.

3.3. Schedule Implementation

Most schedules work out of the box, because they do not need to access any runtime specific function. Halide generates them by altering the source code as operations such as splitting and unrolling are just modification of the loops of the pipeline. Even the

vectorize schedule does not need any specific instructions, Halide will rewrite the schedule as if the pipeline was manipulating a vector even if the hardware target does not support SIMD instructions. To use this schedule fully, a hardware specific modification would be needed. Memory access and thread task distribution on the cluster have to be overwritten as they use specific runtime functions.

3.3.1. Modification to the PULP runtime

The missing Halide functions need to be accessible to the PULP runtime. To do so, we created a new file in the PULP runtime (`halide_api.c`). This file contains all the API functions required to run Halide on HERO.

```
int halide_do_par_for(void *user_context,halide_task_t task,
    int min, int size, uint8_t *closure) {
    // Mount the cluster
    rt_cluster_mount(1, 0, 0, NULL);

    unsigned arguments[4];
    arguments[0] = (unsigned)user_context;
    arguments[1] = (unsigned)size;
    arguments[2] = (unsigned)closure;
    arguments[3] = (unsigned)task;

    // Dispatch the task to the cluster
    rt_team_fork(0, pulp_do_halide_par_for_fork, arguments);

    // Unmount the cluster
    rt_cluster_mount(0, 0, 0, NULL);

    return 0;
}
```

Listing 3.1: The `halide_do_par_for` function.

The Listing 3.1 shows the full source code of the `halide_do_par_for` function. This function is only called once during the execution of a parallel schedule. On the hardware simulator, only the first core of the cluster executes it. The `size` argument of the function determines the number of threads to create, and will be used by the cores to determine which task to execute. This function creates the thread pool for the parallel execution of the pipeline. As HERO does not have a standard way of managing threads, we had to overwrite this function. The `rt_cluster_mount` is called to prepare the cluster before distributing the tasks. The `argument` structure packs all the data about the tasks: the user context, the number of threads, and the starting point of the execution. `rt_team_fork` will then create a team of workers which will all execute the same function: `halide_do_par_for_fork`. The first argument of `rt_team_fork` indicates how many fork the cluster will do, if it is set to zero, the cluster will reuse the last number of threads (which is by default eight).

```
void pulp_do_halide_par_for_fork(void *arg) {
  unsigned *arguments = (unsigned *)arg;

  void *user_context = (void *)arguments[0];
  unsigned task_num = arguments[1];
  uint8_t *closure = (uint8_t *)arguments[2];
  halide_task_t task = (halide_task_t)arguments[3];

  for (unsigned core_id = rt_core_id(); core_id < task_num; core_id += (
      int)&__rt_nb_pe) {
    task(user_context, core_id, closure);
  }

}
```

Listing 3.2: The `halide_do_par_for_fork` function.

The source code of this function is shown in listing 3.2, every worker iterates over the task queue, and executes only the task they have been assigned. The assignment is done by comparing the task identifier with the worker's core identifier, if `core_id = task_id % nb_cores`, then the task will be executed by the worker. The first core then wait for all workers to complete before shutting turning off the cluster and continuing the excution of the application.

This distribution scheme has the advantage of being easy to implement and splitting the task is done in a way that facilitates debugging if needed. But, it can't adapt if one core is delayed or its task takes longer to compute. So if one core has to execute some tasks which are two or three times longer to execute, all the other core will wait for this core to compile. It may be interesting to dynamically reschedule the tasks depending on the free cores.

Chapter 4

Results

4.1. Test Setup

To benchmark Halide, I used a simple matrix multiplication program. We ported the pipeline code from the example in the Halide repository and created the HERO application around it. We chose to benchmark a matrix multiplication as it is a common workflow in signal processing.

```
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 4.1: Matrix Multiplication Pipeline.

The Listing 4.1 shows the full algorithm implementation. We first defined two images parameters: `A` and `B`. and initialize the two main variables `x` and `y`. Then we create a reduction domain, this object defines a domain over which to iterate. This way we can easily bound the sum for each coefficient. Then we compute each coefficient with the function `matrix_mul(x,y)`. The mathematical expression for each coefficient is : $c_{i,j} = \sum_{k=0}^{k=n} a_{ik} * b_{kj}$. We can express is quite similarly using Halide, the program will iterate over the variable `k` repeating the same operation (here a multiplication and accumulation). Then we put the final value inside the output buffer.

To have an idea of the performance, we ran a similar application using OpenMP, and compared the total execution time of both applications. To ensure that the results were correct, the output was precomputed and we compared it to the output matrix. For both applications, the matrices were randomly generated and stored inside the L1 cache. The execution time was measured using `hero_get_clk_counter()` and `hero_reset_clk_counter()`. These two functions respectively reset a cycle counter and return the counter's value when called. Both applications were benchmarked on the hardware simulator.

4.2. Halide Results

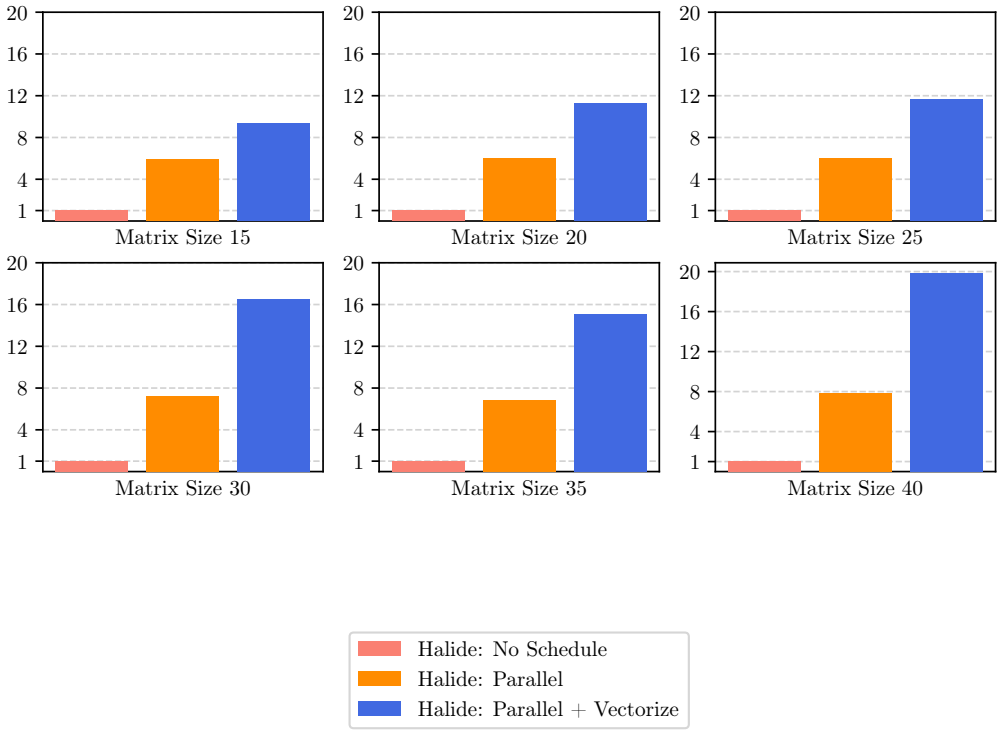


Figure 4.1.: Halide results relative to Halide base schedule performance.

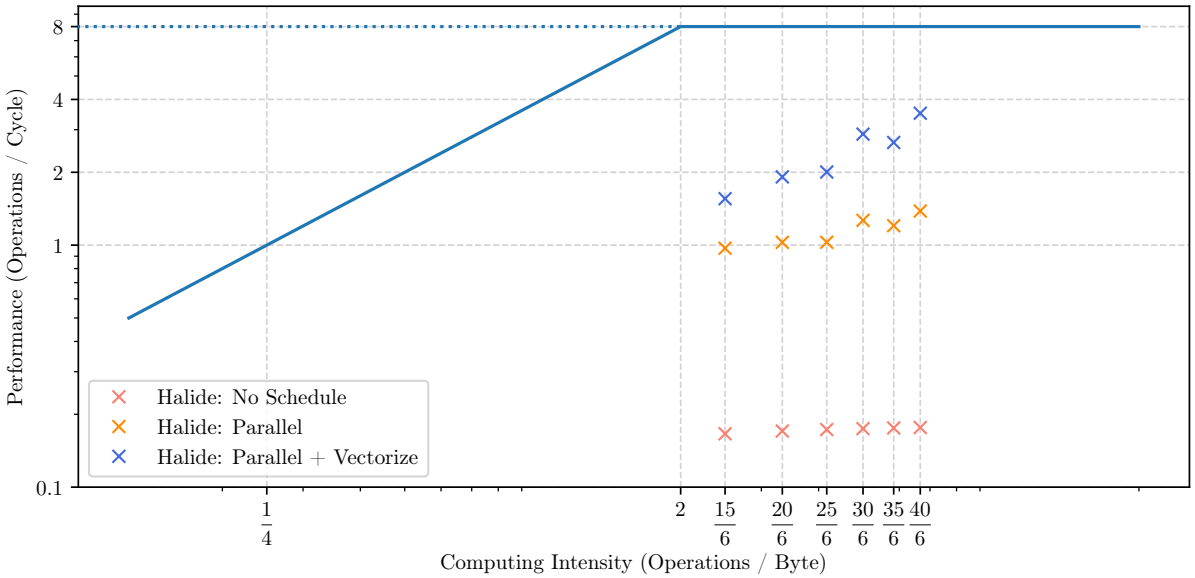


Figure 4.2.: Roofline plot for Halide.

Schedule	15x15	20x20	25x25
Halide: No Schedule	40628 (0.166)	93818 (0.171)	180686 (0.173)
Halide: Parallel	6950 (0.971)	15585 (1.027)	30413 (1.028)
Halide: Parallel + Vectorize	4339 (1.556)	8358 (1.914)	15585 (2.005)

Schedule	30x30	35x35	40x40
Halide: No Schedule	309426 (0.175)	488316 (0.176)	725606 (0.176)
Halide: Parallel	42659 (1.266)	71279 (1.203)	92536 (1.383)
Halide: Parallel + Vectorize	18776 (2.876)	32295 (2.655)	36487 (3.508)

Table 4.1.: Benchmark results in number of cycles and operation per cycles for Halide.

The figures 4.1 and 4.2 show the results of Halide for different matrix sizes ranging from 15 to 40. The benchmarks used three different schedules: the default one, with no parallelization, the parallel schedule along the y-axis, and one schedule which combines a parallel schedule with a vectorize one, the third one is a more optimized schedule, we will talk about it in greater detail in section Optimizing the schedule of the matrix multiplication.

The bar plot 4.1, compares the gain in performance for each schedule relative to the base Halide schedule. When parallelizing the multiplication, we observe between five

and eight times better performances. When the matrix size is a multiple of eight (the number of cores), we get the best scaling possible. This is due to the way Halide handle task distribution, if the number of tasks is not a multiple of the number of cores, some core will stay inactive waiting for the other to complete their tasks. For example, for a matrix of size twenty-five, during three rounds, every core will compute three coefficients, and the first core will compute four coefficients. So during the last round, only one core will be working, reducing the scaling of the program. If we compute the average number of coefficient computed per round, for a matrix of size twenty-five, we get: $\frac{25}{4} = 6.25$. This average matches the increase of performance we observed in our benchmarks. The parallel schedule works best when the number of tasks can be divided by the number of available cores.

Figure 4.2 shows the performance of Halide compared to an ideal scenario. In this case, an ideal core can achieve one arithmetic operation per cycle (which can be done by a PULP cluster as the multiplications can be pipelined). So the total number of operations needed to complete one matrix multiplication is: $2n^3$. The compute intensity of the program represents the number of operations we execute on each byte. We execute $2n^3$ operations, and $3n^2$ memory load / write of 32-bit integers, the overall computational intensity of the program is: $\frac{2n^3}{12n^2} = \frac{n}{6}$.

The default schedule performs identically independently of the matrix size, but the performance of the parallel schedule and the vectorize one increase with the size. This is due to the overlap in memory access, as every core can read four bytes in the L1 cache every cycle, by overlapping the requests, we decrease the time loss due to the memory latency.

4.3. Comparison with an already working toolchain: OpenMP.

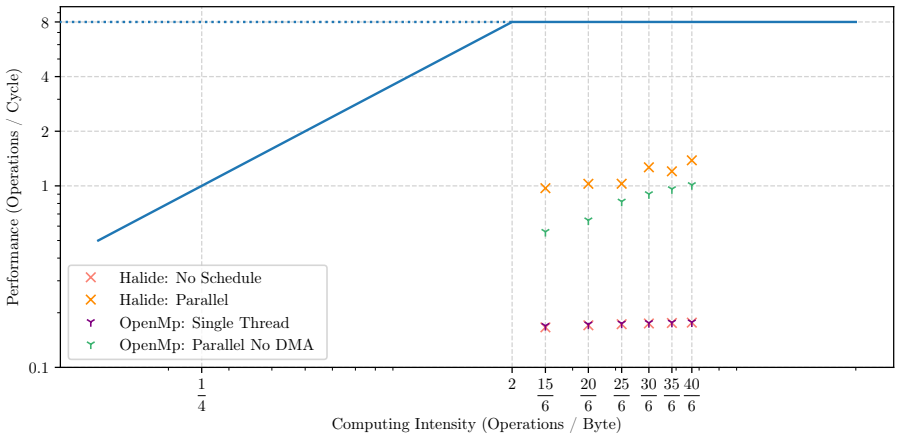


Figure 4.3.: Roofline plot for Halide

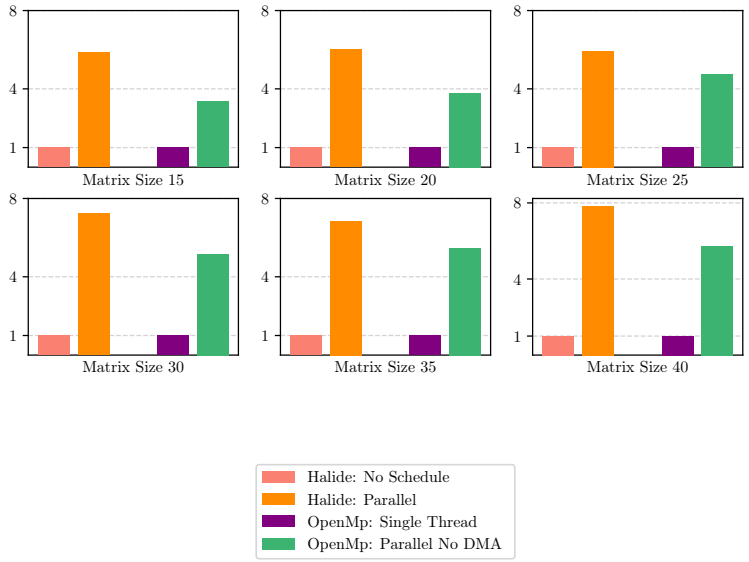


Figure 4.4.: Halide results relative to Halide base schedule performance.

Schedule	15x15	20x20	25x25
Halide: No Schedule	40628 (0.166)	93818 (0.171)	180686 (0.173)
Halide: Parallel	6950 (0.971)	15585 (1.027)	30413 (1.028)
OpenMp: Single Thread	39820 (0.17)	92650 (0.173)	179030 (0.175)
OpenMp: Parallel	12079 (0.559)	24750 (0.646)	38090 (0.82)

Schedule	30x30	35x35	40x40
Halide: No Schedule	309426 (0.175)	488316 (0.176)	725606 (0.176)
Halide: Parallel	42659 (1.266)	71279 (1.203)	92536 (1.383)
OpenMp: Single Thread	307210 (0.176)	485440 (0.177)	721970 (0.177)
OpenMp: Parallel	59887 (0.902)	89283 (0.96)	126523 (1.012)

Table 4.2.: Benchmark results in number of operations (operations per cycle) for Halide and OpenMP.

Currently, the cluster is handled by OpenMP, we decided to compare both solutions to see how Halide performed against it. Table 4.2 presents the results of both schedules. Both applications perform the same using only a single core. The execution time difference is less than four thousand cycles on forty by forty matrices (0.5% of the total execution time).

On the multithreaded run, Halide managed to outperform OpenMP, achieving 40% more operations per cycles on thirty by thirty matrices and 35 % more performance on forty by forty matrices. It seems that OpenMP catch up to Halide on bigger matrices, but we did not test any further as the goal of the project was not to determine which API was the best but to see if we could achieve similar performance to OpenMP with Halide.

4.3.1. Optimizing the schedule of the matrix multiplication

We first compared the performance of every basic schedule on twenty by twenty matrices, unsurprisingly, the parallel schedule performed the best, and the vectorized one was close behind. Every other schedules (unroll, tile, fused, split) performed slightly worse than the base schedule because they all added additional computations or additional jumps. The vectorize schedule performed twice as good as the default schedule even though LLVM do not support the PULP SIMD extension. This jump in performance is due to data reutilisation. On the split, unroll, and tile schedule, if we look at the generated assembly, the code does not reuse any already loaded coefficient, every value loaded from the cache is used only once. Moreover, splitting the code introduce even more loops, which may take more time to execute due to the jump instructions.

On the other hand, the `vectorize` schedule loads one coefficient of the first matrix, and a k coefficients of the second one (where k is the vector length specified when programming the schedule) to compute at the same time k coefficients of the output. To compute k coefficient, we will do $n(k+1)$ memory load, with k memory store. We can compute all the coefficients in: $\frac{n^2}{k}$ iterations, and $n^3\frac{k+1}{k} + n^2 \approx n^3 + n^2$ memory load (instead of: $n^2.(2n+1) = 2n^3 + n^2$ memory operations). But we can't increase k as much as we want as we only have a limited amount of registers available. To see which vector length was the best, We exhaustively tried every vector length possible on twenty by twenty matrices. The schedule used to obtain the results shown by Figure **??**, are achieved using the schedule described in Listing 4.2

```
out.parallel(y);
out.vectorize(x, k);
```

Listing 4.2: Schedule using Parallel and Vectorize

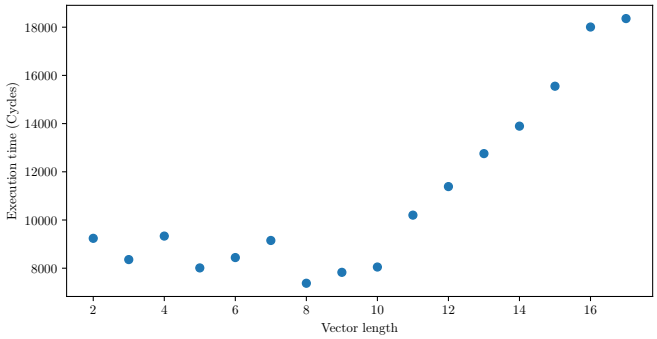


Figure 4.5.: Impact of the vectorization factor on the performance of the application.

We can see on Figure 4.5, that the execution time starts around 9000 cycles for a vector length of two, and decrease to 7300 cycles for a vector length of eight. When the vector length becomes too large, the execution time increases again, because we only have a few registers to work with, and Halide is forced to store some values into the cache when the length is too large.

If we compare our optimized schedule with the two other schedules tested, we get a significant jump in performance. In Figure **??** we can see a difference in performance ranging form ten times to twenty times the single thread performance of Halide. And compared to the multithreaded application, the gains are still significant, as we achieve between 0.5 more operations per cycle up to 2.125 more operations per cycle.

This may not be the most optimal schedule, and for even bigger matrices, more optimal solutions may be found, but the process to get to this schedule was rudimentary, and

without having to change the algorithm, it is really easy to experiment with different schedules and come up with one that performs better.

Chapter 5

Conclusion

5.1. Conclusion

The goal of the project was to port Halide, an image processing language on HERO, and test the language on the hardware simulator. We first added Halide to the HERO repository and automated the build process with the already available tools, then we worked on porting Halide. The porting process was not straight forward as some modifications had to be done on the PULP runtime. The compilation process of OpenMP application had to be changed to enable Halide support. We then tested the implementation with a simple matrix multiplication kernel. We ran the same application on OpenMP to compare the performance of both implementation on this particular workload. The results were on par with OpenMP, and with some optimizations, it is possible to find a more optimized schedule easily. OpenMP will still outperform Halide in most of the workflow, as Halide was designed in the first place for image processing.

During this project, Halide showed promising results, with its ease of programming and the performance of the code. A port of Halide on the full HERO platform would definitely be interesting for signal processing applications on the HERO platform.

5.2. Future Work

5.2.1. Cleaning

Currently, Halide is being ported on another branch and has not been tested, before merging the implementation with the main branch, further testing needs to be done to guarantee that the modified build of LLVM did not break any other toolchains. The branch requires in-depth testing before merging it to the main repository.

5.2.2. Porting Halide to the full platform

As an extension of the project, we tried to port Halide on the hardware platform. We tried two different approaches, one using the code exporting feature of Halide and the second one using the distribution instructions of OpenMP. For the first method, we exported the pipeline into a C file and tried to import it in our application. This method didn't work as some part of the generated code use C++ and not C. Some headers are absent from the HERO SDK, so using this method require further investigation to determine which header are missing and why. The second method was to keep the pipeline as an object file, but adding a `#pragma omp target device(BIGPULP_MEMCPY)` instruction before the pipeline call. Then the goal was to link the precompiled object file during compilation and create a heterogeneous application. But we did not have enough time to figure out how to compile heterogeneous applications using Halide.

Appendix A

Task Description

A.1. Introduction

Heterogeneous systems combine a general-purpose host processor with domain-specific Programmable Many-Core Accelerators (PMCAs). Such systems are highly versatile, due to their host processor capabilities, while having high performance and energy efficiency through their PMCAs. HERO is a FPGA-based research platform developed at IIS that combines a PMCA composed by RISC-V cores, implemented as soft cores on an FPGA fabric, with a hard ARM Cortex-A multicore host processor.

Heterogeneous systems have a complex programming model, which lead to significant effort to develop tools to retain a high programmer productivity. Halide is domain specific programming language designed to write fast image processing algorithms. More specifically, it is a C++ dialect with a functional programming paradigm. It's aim is to separate the function applied to the image (pipeline), and the sequence in which the algorithm is executed (schedule). For example, the schedule encompasses how the algorithm is parallelized, if the image is tiled, processed in column or row major order, if solutions required by multiple threads are shared or recomputed, if parts of the computation is offloaded to an accelerator, and so on. This allows a programmer to write a functional description of the image processing algorithm and then explore ways of scheduling the execution with only a couple of lines of code, and without modifying the algorithm. Furthermore, the same algorithm can be run efficiently on multiple different architectures by only changing the schedule. To have Halide generate efficient code, the specific architecture requires to have an efficient Halide runtime implementation, and good compiler support, as Halide is tightly coupled with the compiler.

A.2. Project description

The goal of this project is to bring up Halide on HERO, using Ariane, a 64-bit RV64GC core, as a host processor. Ariane would manage Halide's frontend, while the image processing tasks would execute on 32-bit cores in the cluster. The final goal of this thesis is to have Halide programmed image processing kernels running on an HERO system implemented on an FPGA.

The project can be done by as one or two semester thesis. The project consists of three parts:

1. Familiarizing with the Halide language and the architecture of HERO (~2 person weeks).

2. Add a RISC-V target to Halide's frontend (~3 person weeks).

3. Test up the Halide environment on an FPGA with a set of custom image processing kernels (~1 person week)

4. Documentation and report writing (~1 person week)

A.3. Required skills

To work on this project, you will need:

- to have worked in the past with at least one RTL language (SystemVerilog or Verilog or VHDL). Having followed the VLSI 1 course is recommended.
- to have prior knowlegde of the C++ programming language
- to have prior knowledge of hardware design and computer architecture
- to be motivated to work hard on a super cool open-source project

Status: In progress

- Student: Pierre-Hugues Blelly
- Supervision: Matheus Cavalcante, Samuel Riedel, Andreas Kurth

Professor

Luca Benini

A.3.1. Meetings & Presentations

The students and advisor(s) agree on weekly meetings to discuss all relevant decisions and decide on how to proceed. Of course, additional meetings can be organized to address urgent issues.

Around the middle of the project there is a design review, where senior members of the lab review your work (bring all the relevant information, such as prelim. specifications, block diagrams, synthesis reports, testing strategy, ...) to make sure everything is on track and decide whether further support is necessary. They also make the definite decision on whether the chip is actually manufactured (no reason to worry, if the project is on track) and whether more chip area, a different package, ... is provided. For more details refer to (1).

At the end of the project, you have to present/defend your work during a 15 min. presentation and 5 min. of discussion as part of the IIS Colloquium.

A.3.2. References

1. Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, Luca Benini. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. CARRV' 2017. link

2. Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, Frédo Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. SIGGRAPH 2012. link

Appendix B

Declaration of Originality

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

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.

Title of work (in block letters):

IMPLEMENTATION OF AN HETEORGENEOUS SYSTEM ON AN FPGA

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
BLELLY	PIERRE-HUGUES

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Toulouse 25/05/2020	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Bibliography

[1] P. J.Basford, S. J.Johnston, C. S.Perkins, T. Garnock-Jones, F. P. Tso, D. Pezaros, R. D.Mullins, E. Yoneki, J. Singer, and S. J.Coxa, "Performance analysis of single board computer clusters," 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X1833142X

[2] Wikipedia, "Crazyflie 2.0 - wikipedia," 2020, [Online; accessed 27-May-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Crazyflie_2.0

[3] D. Palossi, A. Loquercio, F. Cont, E. Flamand, D. Scaramuzza, and L. Benini, "A 64-mW DNN-Based Visual Navigation Engine for Autonomous Nano-Drones," 2019. [Online]. Available: https://arxiv.org/abs/1805.01831

[4] A. Venkat and D. M. Tullsen, "Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor," *ICSA*, pp. 121–132, 2014.

[5] A. Shan, "big.little processing - arm," 2012, [Online; accessed 17-May-2020]. [Online]. Available: https://web.archive.org/web/20121022055646/http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php

[6] A. Kurth, P. Vogel, A. Capotondi, and L. Benini, "HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA," 2017. [Online]. Available: https://arxiv.org/abs/1712.06497

[7] OpenMP, "OpenMP FAQ - OpenMP," 2020, [Online; accessed 24-May-2020]. [Online]. Available: https://www.openmp.org/about/openmp-faq/

[8] K. Wolters, "Software Stack for the First Fully Open-Source Heterogeneous SoC," 2019, master Thesis.

[9] OpenMp, "Home - OpenMp," 2020, [Online; accessed 17-May-2020]. [Online]. Available: https://www.openmp.org/

[10] OpenMP, "OpenMP Compilers & Tools - OpenMP," 2020, [Online; accessed 24-May-2020]. [Online]. Available: https://www.openmp.org/resources/openmp-compilers-tools/

[11] C. Barnes, A. Adams, S. Paris, J. M. Ragan-Kelley, F. Durand, and S. P. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," 2013. [Online]. Available: https://dspace.mit.edu/handle/1721.1/85943

[12] EPI, "European processor initiative," 2019, [Online; accessed 28-May-2020]. [Online]. Available: https://www.european-processor-initiative.eu/?p=unsubscribe

[13] Halide, "Halide," 2020, [Online; accessed 17-May-2020]. [Online]. Available: https://www.halide-lang.org/

[14] ——, "Debugging tips · halide/halide wiki," 2020, [Online; accessed 24-May-2020]. [Online]. Available: https://github.com/halide/Halide/wiki/Debugging-Tips

[15] PULP, "PULP Platform," 2020, [Online; accessed 17-May-2020]. [Online]. Available: https://pulp-platform.org

[16] Hero, "HERO: The Open Heterogeneous Research Platform," 2020, [Online; accessed 17-May-2020]. [Online]. Available: https://pulp-platform.org/hero.html

[17] Wikipedia, "Heterogeneous computing - Wikipedia," 2020, [Online; accessed 17-May-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Heterogeneous_computing

[18] ——, "OpenMP - Wikipedia," 2020, [Online; accessed 24-May-2020]. [Online]. Available: https://en.wikipedia.org/wiki/OpenMP

[19] A. Shan, "Heterogeneous Processing: a Strategy for Augmenting Moore's Law | Linux Journal," 2006, [Online; accessed 17-May-2020]. [Online]. Available: https://www.linuxjournal.com/article/8368