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

OpenMP . . . . Open Multi-Processing

PMCA . . . . . Programmable Many Core Accelerator

PULP . . . . . . Parallel Ultra Low Power

RTL . . . . Register Transfert 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.

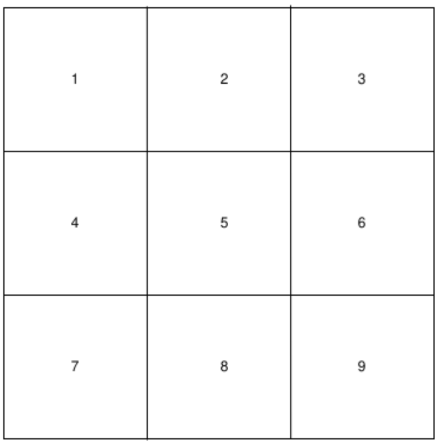
2.2.2. Debugging Options

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.



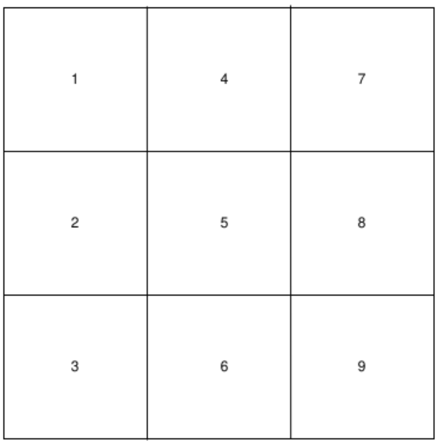

```
# Generated loops
for(int y = 0; y < 3; y++){
  for(int x = 0; x < 3; x++){
    do something on (x,y)
  }
}
```

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



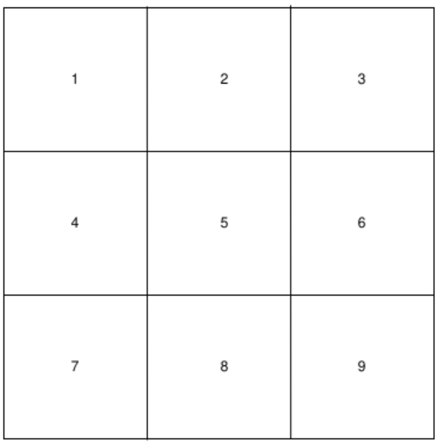

```
  pipeline.reorder(y,x);

# 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



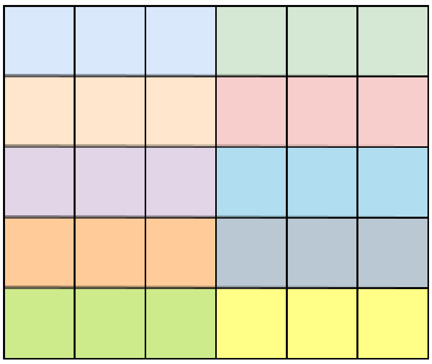

```
  pipeline.fuse(x,y,xy);

# 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



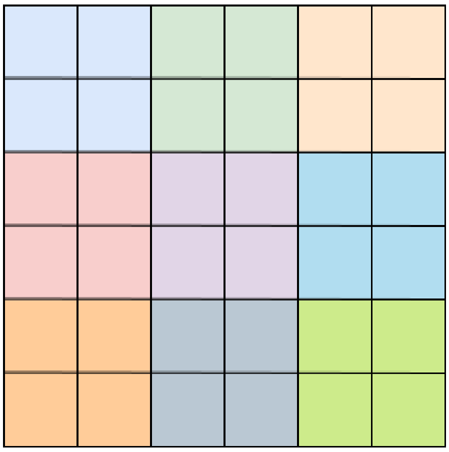
```
pipeline.split(x,x_o,x_i, 3);

# 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



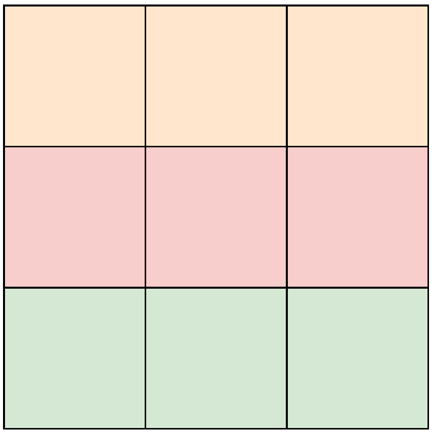
```
pipeline.split(x,y,x_o,y_o,x_i,y_i,2,2);


# 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

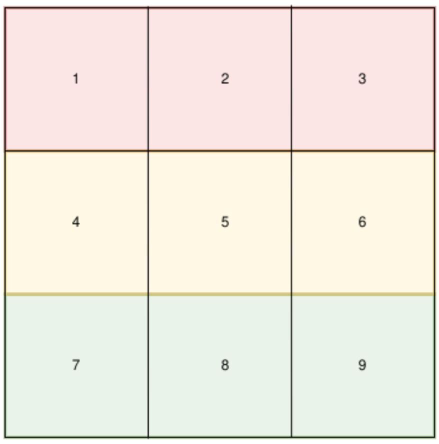


```
# Generated loops
pipeline.unroll(x,3);
for(int y = 0; y < 4; y++){
  do something on (0,y)
  do something on (1,y)
  do something on (2,y)
}
```

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 rely on the LLVM library to generate compiled 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 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. The hardware simulation runs an Register Transfert Level (RTL) simulation of a PULP cluster configured with eight cores. This platform is easier to work on as it does not require any specific hardware (FPGA) and uses one of the cluster's core as the host of the system.

3.2. Compilation Workflow

```
halide-examples/
├── common/
│   └── defaultHalide.mk
└── myApp/
    ├── main.c
    ├── Makefile
    └── lib/
        └── halidePipeline.cpp
```

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 used during the compilation of the application.

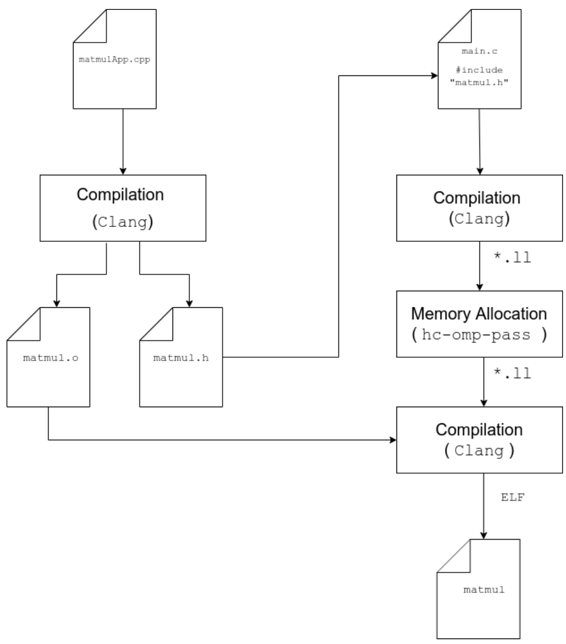


Figure 3.2.: Compilation Workflow for an Halide Application.

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 pipeline into a

RISC-V object file and generate 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 is a 64-bit RISC-V 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 uses 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 keep them as-is for now, 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 don't 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 doesn't need any specific instructions, Halide will rewrite the schedule as if it was manipulating a vector even if the hardware target don't support SIMD instructions. 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 kernel (`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 called when the pipeline uses the parallel schedule. 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 describes all the data about the tasks such as the number of tasks or the function to execute we want to execute. `rt_team_fork` will then create a team of workers which will all execute the same function: `halide_do_par_for_fork`.

```
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. Finally, once every worker completes, the cluster is turned off.

Chapter 4

Results

4.1. Test Setup

To test Halide, I used two applications. The first one was a simple gradient example 2.1, and the second one a matrix multiplication pipeline that I took in the Halide repository and then adapted to be used in a hero application. The matrix example is more interesting, as it represents 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 4.1: Matrix Multiplication Pipeline.

The Listing 4.1 shows the full algorithm implementation. The code is straight forward and is pretty close to the mathematical expression of the operation.

I tested the Halide implementation using two applications, first the simple pipeline described in the listing 2.1 to check if everything was working and then with a matrix multiplication pipeline to be closer to a real scenario. Once I checked that Halide was working correctly. I ran a similar matrix multiplication program using OpenMP to compare the parallelisation performance of Halide against the current toolchain available on

HERO. To check the output of the Halide application, I compared the output of the pipeline to the precomputed result. During all Halide benchmarks, the two input matrices were randomly generated using a python script, as the multiplication on the PULP cluster always take two cycles, the content of the matrices does not impact the execution time of the programs. To get comparable results for both applications, stored the matrices in the L1 cache of HERO to reduce the memory latency. To measure the performance of both applications (on Halide and on OpenMP), I used two functions of the HERO runtime: `hero_reset_clk_counter()` and `hero_get_clk_counter()`. These two functions respectively reset a cycle counter and return the counter's value when called. As their execution is short, we can get cycle accurate result for our benchmarks. The benchmarks were executed on the hardware simulator, as I did not have time to make the heterogenous compilation work.

4.2. Halide Results

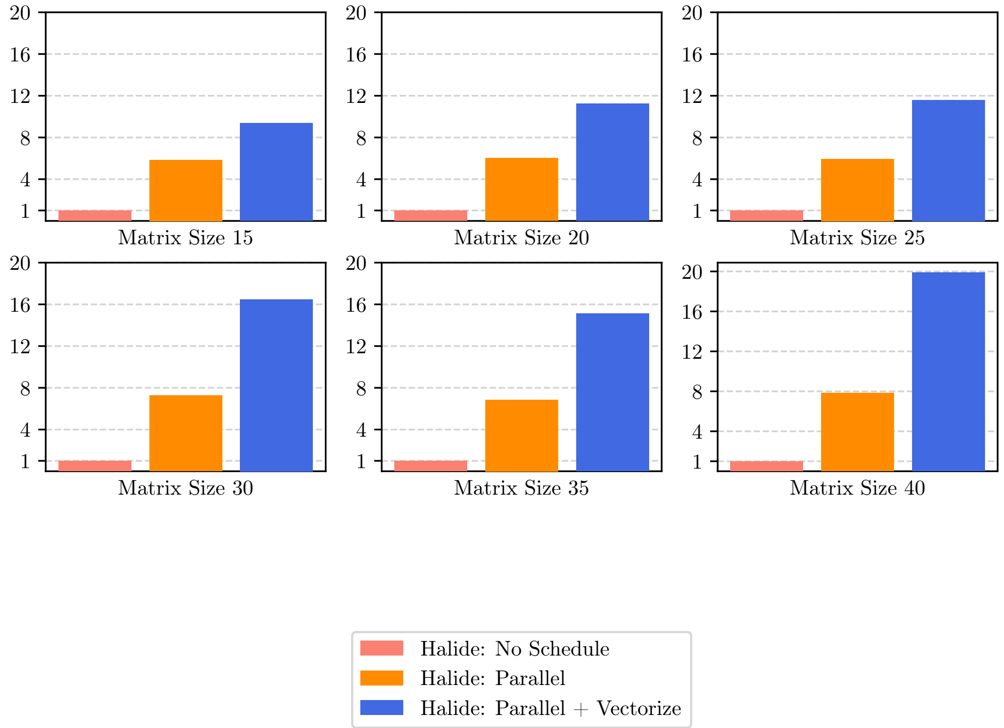


Figure 4.1.: Halide results relative to Halide base schedule performance.

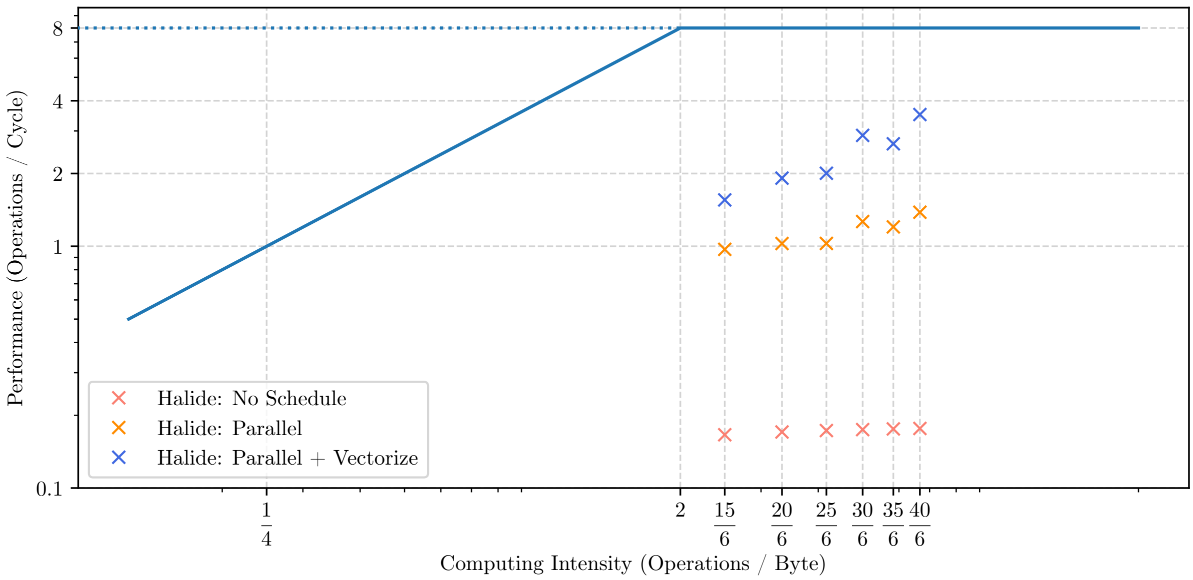


Figure 4.2.: Roofline plot for Halide.

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 were done using three different schedule: the default one, with no parallelisation, one schedule with parallelisation along the y axis, and one schedule which combines a parallel schedule with a vectorize one. On figure 4.1, we can see that the paralellisation instruction is efficient as this schedule performs between five and eight times better than the default schedule.

Figure 4.2 show the performance of Halide compared to an ideal scenario. As one matrix multiplication can be done in $2n^3$ arithmetic operations, in the best scenario, each core achieve one arithmetic operation per cycle, so we can get up to eight operations per cycle.

For the default schedule, we can see that the overall performance stays consant and is not dependant on the matrix size. But for the two other schedule, the performance increase when we increase the matrix size. This can be explained by the core utilisation, as every core only executes the task when: `core_id == task_id % n_cores`, the overall core utilization is lower for smaller matrices when the matrix size is not a multiple of the number of cores. For a matrix of size twenty, four cores will execute three tasks, and four cores will execute two tasks. But for bigger matrices, core will stay inactive for a smaller fraction of the total execution time, increasing the overall performance of the system.

4.3. Comparison with an already working toolchain: OpenMP.

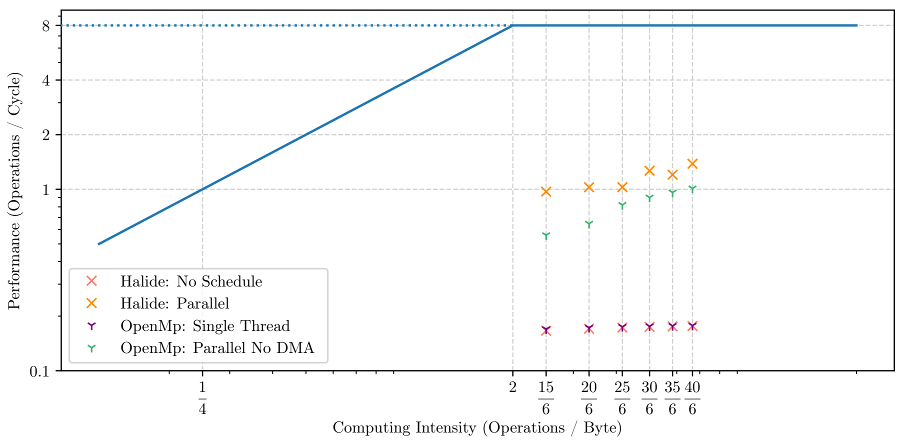


Figure 4.3.: Roofline plot for Halide

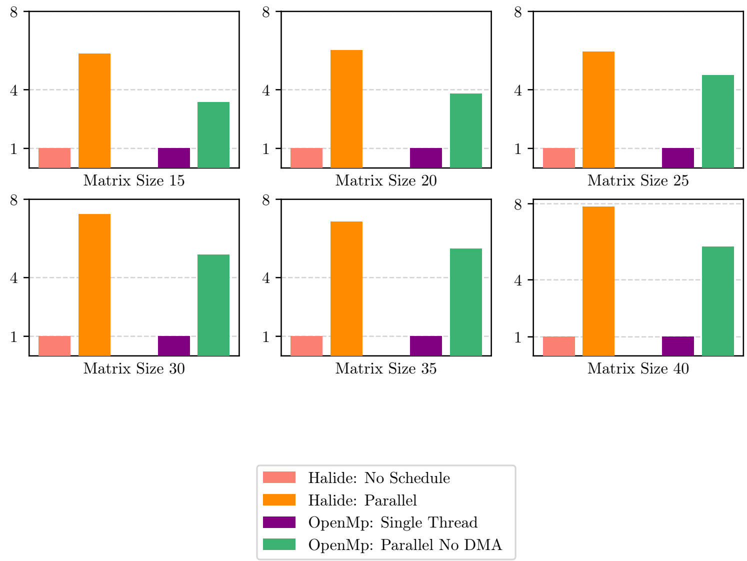


Figure 4.4.: Halide results relative to Halide base schedule performance.

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 No DMA	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 No DMA	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

As HERO already have a working workflow to distribute computation on the PULP cluster using OpenMP, we can see how Halide compare to this API on a simple parallel schedule. First, we can see that both implementation of the matrix multiplication achieve the same parformance when using only a single core, differing only by less than four thousand cycles for fourty by fourty matrices.

When we parallelize the applications accross all the cores, we can see the performance gap increasing. On fourty by fourty matrices, Halide achievs almost eight times the performance of the single threaded program, where OpenMP achievs around six times the performance.

On bigger matrices, Halide seems to scale better than OpenMP. Halide performs worse on some sizes (thirty five and twenty five) because of how the tasks are handled. When the size of the matrices is not a multiple of eight, we don't achieve full core utilization during the last round, reducing the overall performances. On twenty-five by twenty-five matrices, during the last round, only one core is used, achieving on average $\frac{25}{4} = 6.25$ times the performance of the single core program.

On this applicationn, we can see that Halide out performs OpenMP. I then tried to optimize the schedule to see on this application how Halide performs with a more advanced schedule.

4.3.1. Optimizing the schedule of the matrix multiplication

I first compared the performance of every basic schedule except the parallel one to see if they impacted the overall result. Every schedule (unroll, split, tile, fuse) was performing slightly worse than the base one except the vectorize schedule, which had an execution time two or three time smaller than the other ones. This can be explain by data reutilisation, on the split, unroll and tile schedule, if we look at the generated asssembly, the code don't reuse any already loaded coefficient, more over as the program often has more jumps to do, the overall execution time increase. 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 in the schedule) to compute at the same time k coefficients of the output.

To see which vector length was the best, I exhaustively tried every vector length possible on twenty by twenty matrices. The schedule used to obtain the results shown by Figure **??**, are achieved using this schedule: `out.parallel(y).vectorize(x,k);`, where `k` is the vector length.

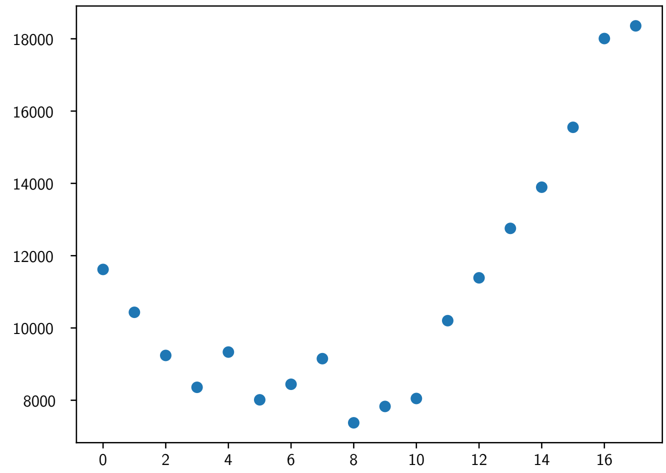


Figure 4.5.: Impact of the vectorization factor on the performance of the application.

Figure **??** shows that increasing the vector length too much only decrease the overall performance, this is due to the limited amount of registers. As RISC-V CPUs have 32 registers, and some of them have dedicated functionnalities, we can't use all of them. So when we increase the vector length, at one point, there are no available registers to store intermediate computations, and the compiler is forced to store these intermediate values in the caches, which take some cycles. On RISC-V CPUs, the sweetspot seems to be a vector length of eight.

I made sure that the two matrices were stored in the L1 cache, to have the best access time possible. To measure the number of cycles needed to run the application, I used two functions available in the hero sdk: `hero_reset_clk_counter()` and `hero_get_clk_counter()`. These functions reset and return the value of a cycle counter. As they only take few assembly instructions, they are useful to get cycles accurate measurements of the execution time. 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. I then experimented with different schedule with Halide to see the maximal performance I could get with this application.

To give the results more meaning, I converted the benchmark data in operations per cycles where one operation can either be an addition or a multiplication, so for a matrix of size n, the number of operations to finish the multiplication is : $2n^3$.

4.4. Comparaison between OpenMP and Halide on the different platforms

Halide base schedule performs similarly from OpenMP single threaded, differing only by a few thousands cycles for the last test (with two matrices of size 35 by 35). The two implementation are prettty similar, and Halide overhead is in this case almost negligeable.

The second schedule I tested on both APIs was the parallel schedule; as parallelisation is the most efficient way to increase performance especially when there is no data dependancy in the pipeline. To have the best possible performance, the API needs to be as small as possible.

The table **??** show the results of both applications, with every size tested, we can see that Halide is in every situation at least .2 operations per cycle faster than OpenMP.

I also tried multiple schedule for Halide to see the best performance HERO could achieve on this benchmark. I tried to combine the parallel schedule with loop unrolling or tiling, but I was only getting worse results due to the additional jumps or the additional computation implied by loop unrolling. When the unrolling factor is not a divisor of the number of loop iteration, Halide shift the final iteration to always compute the same number of element each iteration. This shift forces the pipeline to recompute some output values.

```
out.parallel(y);
out.vectorize(x, 10);
```

Listing 4.2: Schedule using Parallel and Vectorize

The vectorize schedule used with parallel proves to be the most efficient solution, on twenty by twenty matrices, the parallel schedule alone achieved 1.026 operations per cycle, against 2.169 operations per cycle using the schedule 4.2. I then exhaustively tried every vectorization factor possible to see which performed best.

Chapter 5

Conclusion

5.1. Conclusion

The goal of the project was to port Halide, an image processing language on HERO, and run image processing kernel on the test hardware. Some functions were missing on the PULP runtime, so we first added them to the source code. Then we added the Halide source to the HERO project and built the library. The compiling options of LLVM needed to be changed to sucessfully compile Halide. Using the Makefile of the OpenMP applications as a base, I sucessfully compiled Halide applications to run on the RTL simulator. I then tested two applications a gradient and a matrix multiplication to debug the schedules and test if they were working correctly. After that, I ran some benchmarks on different matrix sizes to compare the perfromance of Halide and OpenMP to determine whether Halide could compete with OpenMP or not. I then tried to make the heterogeneous compilation work on the HERO platform with the ARM host but I did not have enough time make it work. I slightly changed the LLVM target to include other object file during linking. But in the end I did not have enough time to make it work on the hardware platform.

Even if I could not finish the project, on the RTL simulator, Halide showed promising results, but it need to be benchmarked more thoroughly to have a better idea of the performances Halide can achieve.

5.2. Future Work

A lot of work needs to be done to merge Halide on the main branch of HERO, The heterogeneous workflow for Halide needs to be fixed as it is impossible right now to distribute code to the PULP cluster from the ARM cluster. The Continuous Integration (CI) is currently not working, which is probably due to the change of options when compiling LLVM, this may also cause compability issues with OpenMP or other components of the toolchain. This branch requires in depth testing before being merged with the main project.

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