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 / Master Project

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Pierre-Hugues BLELLY
pblelly@student.ethz.ch

May 2020

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

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

Acknowledgements

Abstract

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
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	Massachusett Institute of Technology
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing

PCMA	Programmable Many Core Accelerator
PULP	Parallel Ultra Low Power
RISC-V	RISC-V
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 for a power consumption of only 7 Watts [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. Nano drones can now analyze in real-time a video signal and train a neural network for autonomous navigation, at a rate of 281MMAC/s on a power-enveloppe of 45mW [2].

To improve the energy efficiency and the computing power of Ultra Low Power (ULP) systems, new architectures are needed. To keep the power consumption low, an embedded system needs to power it's subsystems only when needed. Autonomous drones [?] use a low-performance microcontroller to manage it coupled with a high-performance ULP cluster of RISC-V (RISC-V) cores for signal processing. System composed by one host processor and one or multiple coprocessors are called heterogeneous systems, they are suited for embedded applications as they can keep a low power consumption while using high-performance accelerator when needed.

This strategy has been used in the System on Chip (SoC) industry by ARM since 2011 [?]. 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 Instruction Set Architecture (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.

Researchers from the University of California [3] tried to leverage the advantages of multiple ISAs. Under heavy design constraints (such as die area or thermal dissipation) heterogeneous systems based on multiple ISAs performed better than the best homogeneous counterpart (in terms of energy efficiency and compute performance).

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) whichc simplify the developement process for GPU accelerators.

HERO [4] is a heterogeneous system developed by the Integrated Systems Laboratory (IIS) of ETH Zürich and the Energy Efficient Embedded Systems (EEES) of the University of Bologna. This platform is composed of a hard multicore ARM 64 Juno SoC (composed of two Cortex A57 and four Cortex A53 cores) and up to eight PULP clusters (composed of eight RI5CY cores [4]), running on an Field Programmable Gate Array (FPGA)(a Xilinx ZYNC ZC706). The PULP cluster is based on the RISC-V ISA, an open source ISA designed to support a wide range of platform from embedded systems to supercomputer. The modularity of the ISA makes it interesting for Programmable Many Core Accelerators (PCMAs).

This platform is designed to "facilitate rapid exploration on all software and hardware layers" [4], and includes a heterogeneous compilation toolchain with support for OpenMP, an API developed to make developement of multi threaded aplications easier [**?**]. 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 specify how the CPUs in the system will interact will each other. These choices will impact the peak performance of the design or its power consumption [3]. The computer architect has to choose how the different PCMAs will interact, how they will share data, maybe extend the existing ISAs to distribute tasks, and so on.

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 PCMAs. Code distribution is handled by the programmer, APIs such as CUDA, OpenCL or OpenMP using function calls tell the compiler how to execute the code and on which PCMA.

1.2. Currently Available Workflow for HERO

Currently, HERO supports OpenMP [5], 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" [6]. This API has been implemented on HERO to easily take advantage of the PULP clusters. The toolchain uses the Clang compiler [7] to compile the applications. HERO uses custom Clang front ends to supports all the available configurations (only the PULP cluster for simulation, with the ARM host CPU 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 directive 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 algorithmn, and complex schedules often impact code readability making them harder to debug

Halide [8] was proposed to explore the idea of separating the algorithm from the schedule. This separation makes testing different schedule 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 loops, unrolls them. The developer can still has the freedom to implement any schedule he wants but without having to change the main algorithm. This programming model is interesting because the developer can quicky implement the algorithm without having to take into account the boundaries of the inputs, and then work on an optimal schedule.

The intermediate variables can be bounded afterwards if needed, and the pricipal 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.

Chapter 2

Background

2.1. Hero

2.2. Halide Language

2.2.1. Programing model

Halide is a functionnal programming language embedded into C++, designed to write high performance image and array-processing code [9]. This language uses a functionnal 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 compute 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 variable `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 wanted. 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 variable together into a single one, or more advanced instructions like parallelization or vectorization are also available. These instructions can be combined as needed to create complex schedules. Section Basic Scheduling Options explain the most important scheduling instructions in greater details.

In the listing 2.2, we can see how we program a schedule.

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 Pipeline 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 computer. 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 primarly to work with different hardware platform, the cross-compilation process has been simplified, and the pipeline can be translated to other languages. Halide support translation to C code, Low Level Virtual Machine (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 system(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 [10].

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 hasn't been inlined, the parameters and the result of the function call will be stored in the trace and printed after the exeuction.

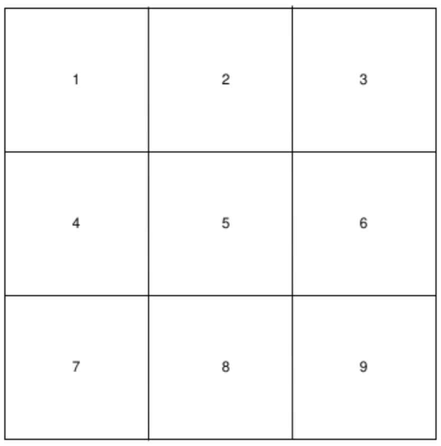
Halide can print more information on the screen during the compilation of the source code by setting the environemental variable `HL_DEBUG_CODEGEN` to 1. Halide will output information about every stages of the compilation and a pseudo code representation of the pipeline loops. Finally, variables and 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.

2.3. Basic Scheduling Options

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

2.3.1. Non Platform Specific Schedule

Default Schedule

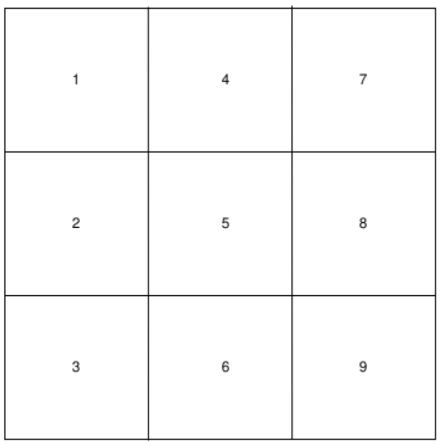


```
for(int y = 0; y < 4; y++){
  for(int x = 0; x < 4; x++){
    do something ...
  }
}
```

Figure 2.1.: Base 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

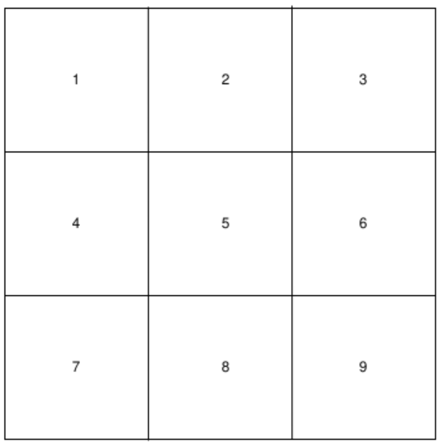


```
  pipeline.reorder(y,x);
for(int x = 0; x < 4; x++){
  for(int y = 0; y < 4; y++){
    do something on
    (x,y)
  }
}
```

Figure 2.2.: Schedule: Reorder

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

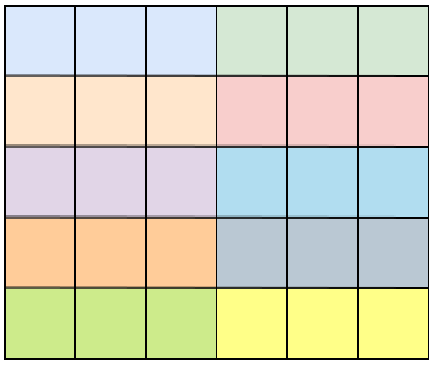


```
  pipeline.fuse(x,y,xy);
for(int xy = 0; xy < 9; xy++){
  do something on
  (xy)
}
```

Figure 2.3.: Schedule: Fused

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

Split

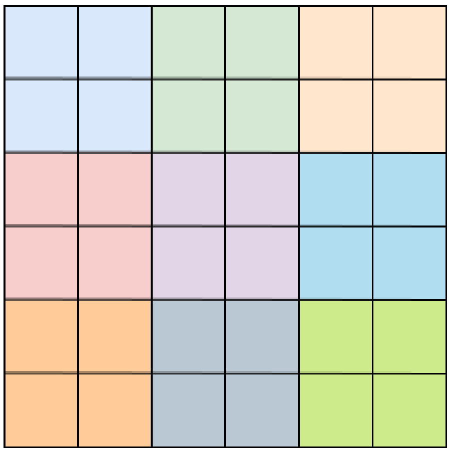


```
pipeline.split(x,x_o,x_i, 3);
for(int y = 0; y < 6; y++){
  for(int x_o = 0; x_o < 3; x_o++){
    for(int x_i = 0; x_i < 4; x_i++){
      do something on
      (x_o * 3 + x_i, y)
    }
  }
}
```

Figure 2.4.: Schedule Split

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

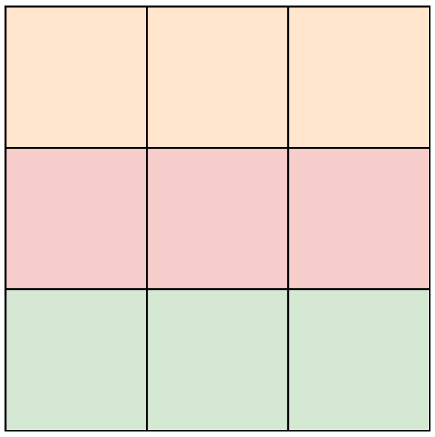


```
pipeline.split(x,y,x_o,y_o,x_i,y_i,2,2);
for(int y_o = 0; y_o < 6; y_o++){
  for(int x_o = 0; x_o < 3; x_o++){
    for(int y_i = 0; y_i < 2; y_i++){
      for(int x_i = 0; x_i < 2; x_i++){
        do something on
        (x_o * 2 + x_i, y_o * 2 + y_i)
      }
    }
  }
}
```

Figure 2.5.: Schedule Tile

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



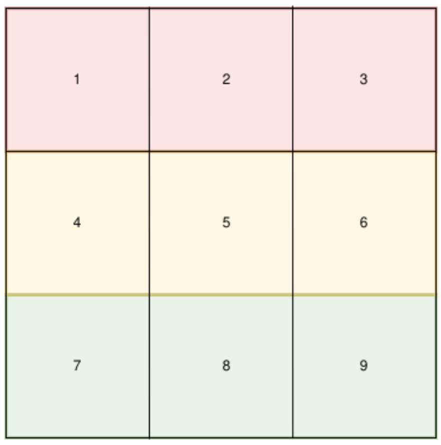
```
pipeline.split(x, x_o,x_i,3);
pipeline.unroll(x_i,3);
for(int y = 0; y < 4; y++){
  do something on (0,y)
  do something on (1,y)
  do something on (2,y)
}
```

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 example 2.6, we first split the x dimension before unrolling as Halide can't unroll a variable if it isn't bounded.

2.3.2. Platform Specific Schedules

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.: Schedule Parallel

The parallel schedule distributes the pipeline to all the available cores. Halide will create for task for each value the variable can take, and these tasks will be executed with the `halide_do_par_for` function. This function has been overwritten on hero to execute on the PULP cluster. In the example 2.7, the code is distributed on three cores, each of them execute a single loop along the y axis.

Vectorize

The goal of this schedule is to setup the code so to make use of the SIMD instructions of the CPU. Currently, llvm doesn't support the vector 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.

2.3.3. Porting Halide to new Platforms

In order to compile Halide, we need to compile llvm with the flag without shared libraries otherwise, Halide won't compile, and with support for the desired targets (x86, RISC-V and 64 bit ARM architecture (AARCH64)).

Using the comment inside the pipeline header file, we can determine which function we need to implement to make Halide work on our target platform. The error messages during the linking phase are also a good source of information to find which function are needed to compile the code.

Currently only the memory allocation functions, the print functions and the task distribution functions are implemented, and they are enough to test basic pipelines such as matrix multiplications or light image modifications. After the implementation we can work on the compilation workflow for hero.

2.4. Compilation Workflow

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

2.5. The full hero platform

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

Chapter 3

Design Implementation

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

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

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

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

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

Listing 3.1: Matrix Multiplication Pipeline

The listing 3.1 shows the full algorithm. The implementation is straight forward and is pretty close to the mathematical expression of the operation.

3.1. Schedule Implementation

Most of the schedules implemented on halide doesn't require any platform specific implementation as they are only unrolling, splitting or swapping loops. During my project I used two platform specific schedules: vectorization and parallelization.

3.1.1. Modification to the PULP runtime

The missing halide functions needs to be accessible to the pulp runtime, we added a new file in the kernel of the runtime: `halide_api.c`. This file contains all the basic function required to run halide on Hero. This files implement Atomic Operations needed for parallelisation, and a all the function needed to try basic schedules.

Currently, only the `parallel()` instruction needs a specific function. The task distribution is done using `halide_do_par_for`. This function initialise the PULP cluster and initialize all the tasks un the queue. These task executes `halide_do_par_for_fork` which is a wrapper around the pipeline function to select which task to execute on which core. To know which task to execute, the core execute the task only if the task number modulo the number of core on the platform is equal to the core id.

Halide doesn't support the RISC-V vector extension, but it can still be used on the platform, on PULP, the code will be reshaped to compute the targetted domain as a single vector. But won't use the vector extension to do so.

Chapter 4

Results

4.1. Test Setup

To compare Halide and OpenMP, I ran a matrix multiplication program coded using OpenMP and Halide. I ran the applications in the hardware simulation, using only the PULP cluster. The matrices for the Halide application were randomly generated and for the OpenMP application I generated them using a predefined patttern. As the multiplication operation always take two cycles, the value inside the matrices doesn't matter as long as they have the same size. 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 : $2n^3$.

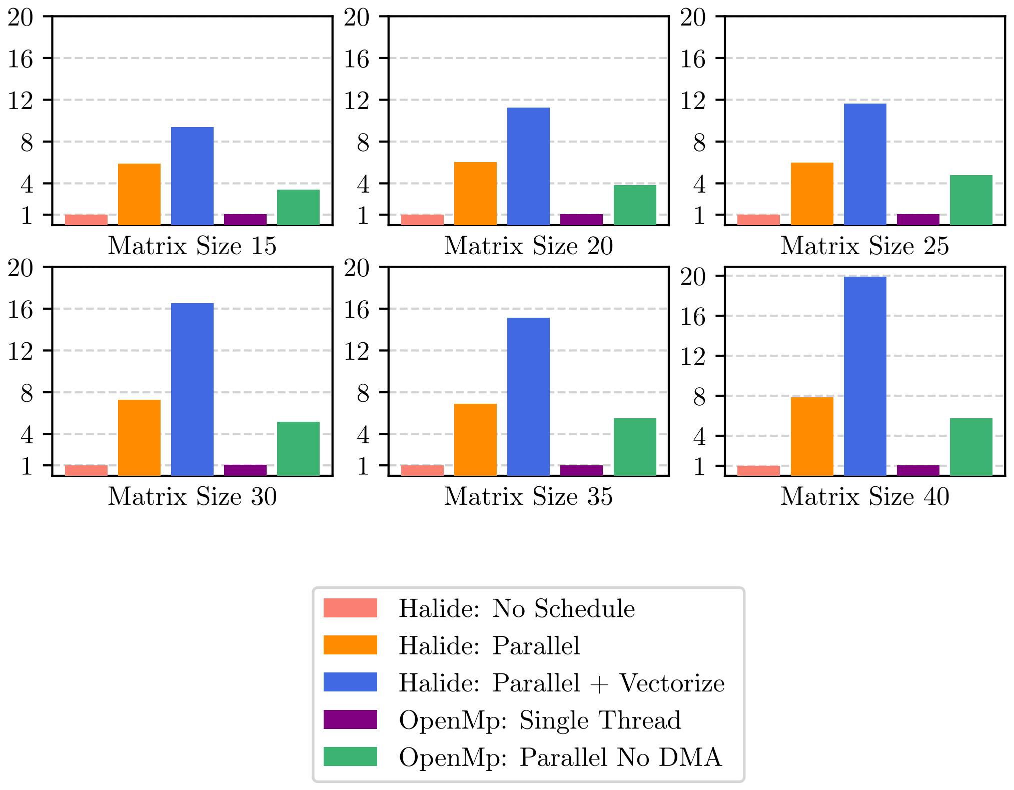


Figure 4.1.: Halide vs OpenMP, relative to halide base schedule performance

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)
Halide: Parallel + Vectorize	4339 (1.556)	8358 (1.914)	15585 (2.005)
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)
Halide: Parallel + Vectorize	18776 (2.876)	32295 (2.655)	36487 (3.508)
OpenMp: Single Thread	307210 (0.176)	485440 (0.177)	721970 (0.177)
OpenMp: Parallel	59887 (0.902)	89283 (0.96)	126523 (1.012)

Table 4.1.: Benchmark results in number of operations (operations / cycle)

4.2. Comparaison between OpenMP and Halide on the different platforms

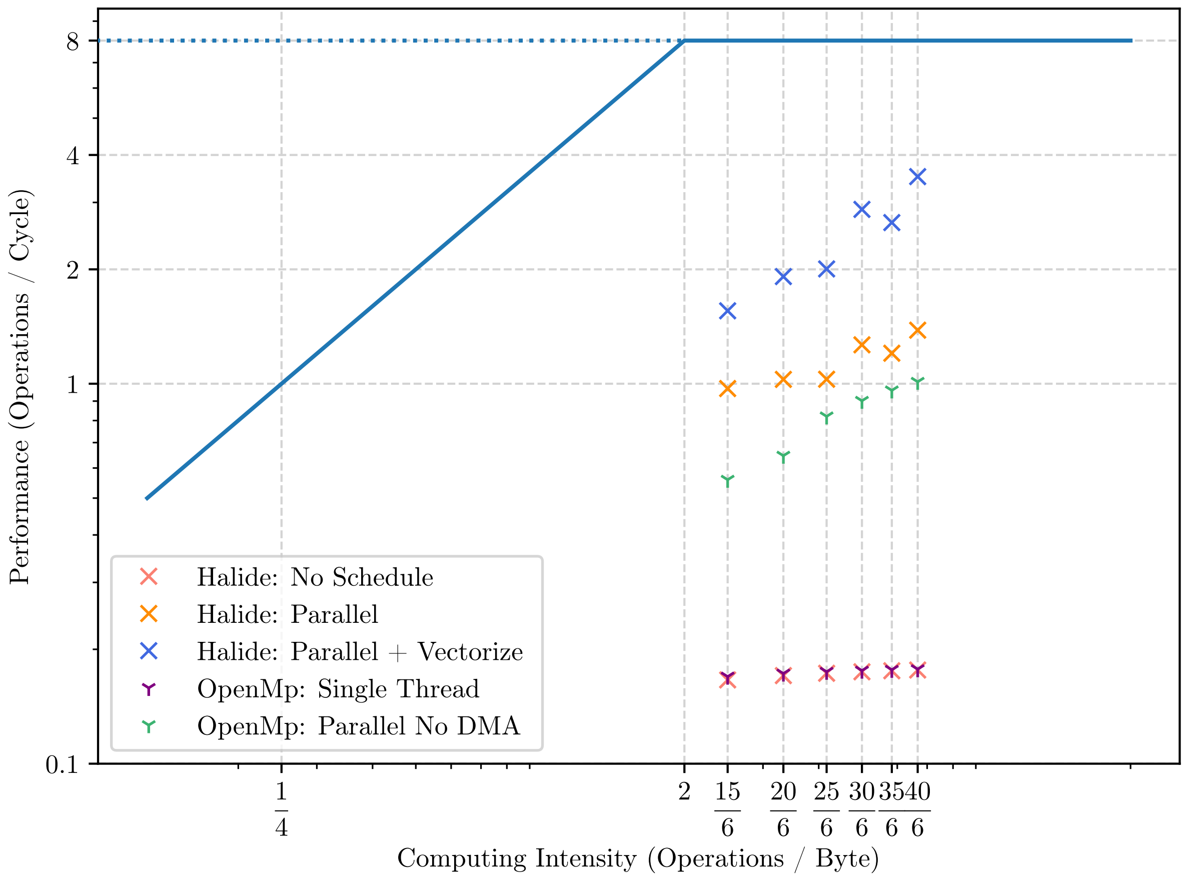


Figure 4.2.: Roofline plot of the benchmark results

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 neglige-able.

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 4.1 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 isn't 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 4.1: 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 4.1. I then exhaustively tried every vectorization factor possible to see which performed best.

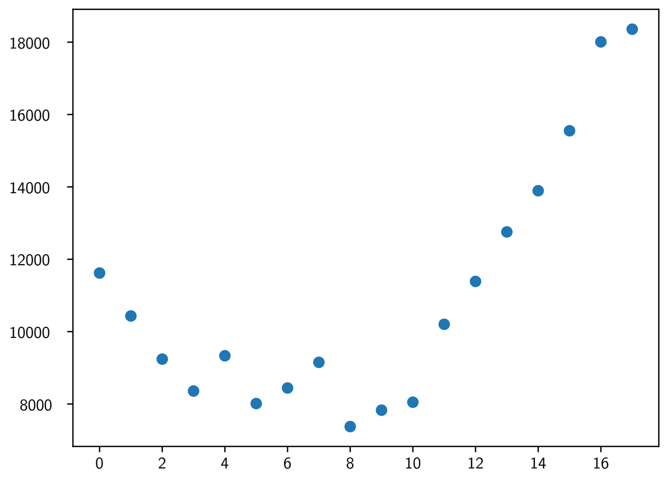


Figure 4.3.: Impact of the vectorization factor on the performance of the application

Chapter 5

Conclusion and Future Work

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

5.1. First Section

5.2. Second Section

Appendix A

Task Description

Include the task description **pdf** you got from your assistant(s) with the `\includepdf` command.

Appendix B

Declaration of Originality

Include the declaration of authorship with the `\includepdf` command (sign it and scan it). For more information about plagiarism, please visit `https://www.ethz.ch/students/en/studies/performance-assessments/plagiarism.html`

- **English version:** `https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/declaration-originality.pdf`
- **German version:** `https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/plagiat-eigenstaendigkeitserklaerung.pdf`

Glossary

Atomic Operations An operation during which a processor can simultaneously read a location and write it in the same bus operation. This prevents any other processor or I/O device from writing or reading memory until the operation is complete..

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] 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

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

[4] 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

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

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

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

[8] 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

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

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

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

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

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

[14] 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