ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and
Electrical Engineering

Spring Semester 2020

LaTeX Report Template

Semester Project / Master Project

Pierre-Hugues BLELLY
pblelly@student.ethz.ch

May 2020

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

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

Acknowledgements

Abstract

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

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

Declaration of Originality

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

Pierre-Hugues BLELLY,
Zurich, May 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Thanks to the smaller nodes of modern lithography technologies and the transistor density we can achieve with them, modern Central Processing Units (CPUs) in embedded systems, as capable enough to be used in datacenters. A single Raspberry Pi 3 has a peak performance of 6 Giga Floating Operations per Second Institute of Technologys (GFLOPs) for a power consumption of only 7 Watts **??**. Embedded systems can take advantage of it 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 under 100mW **??**.

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 **??**, uses a low-performance microcontroller to manage the drone coupled with a high-performance cluster for the signal processing tasks. System using one host processor and one or multiple coprocessors are called heterogeneous systems. They are suited for embedded systems 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 and A15, 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 also tried to leverage the advantages of multiple ISAs, according to this article **??**, 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) which make them capable of heavy computation.

Hero [**?**] is a heterogeneous system developed by the Integrated Systems Laboratory (IIS) of ETHZ 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) and up to eight Parallel Ultra Low Power (PULP) clusters (composed of eight RI5CY cores), running on an Field Programmable Gate Array (FPGA). The PULP cluster is based on the Reduced Instruction Set Computer V(Five) (RISC-V) ISA, this ISA is open source and was 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 ", and includes a heterogeneous compilation toolchain with support for OpenMp.

1.1 Design Issue with heterogeneous systems

Due to their complexity, heterogeneous systems are difficult to design and program **??**.

During their conception, numerous design choices need to be made to make sure that all the CPU in the system will interact will each other, these choices will impact the peak performance of the design or it's power consumption [**?**]. The 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 Open Multi-Processing (OpenMP), specific function calls tell the compiler which code to distribute to which PCMA.

1.2 Currently Available Workflow for Halide

Currently Hero supports OpenMP, which is 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" [**?**]. This API has been implemented on Heterogeneous Embedded Research Platform (HERO) to easily take advantage of the PULP clusters. The toolchain uses Clang, and `clang-offload-bundler` to compile the

applications. Clang uses a custom front head for HERO which 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 which part of the code to offload by specifying the size of the domain. Trying to change the order of multiple loops may cause bugs in the algorithm and so on. This approach requires to spend time retesting the algorithm before trying other schedules.

Halide [**?**] 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 consist of the functional description of the processing kernel, this is the algorithm that will be executed on the array. The second part is the schedule of the pipeline. This schedule describes how the algorithm will be executed on the system. This programming model is interesting because the developer can implement the algorithm without having to take into account the boundaries of the functions or the border effects.

The intermediate variables can be bounded afterward if needed, and the principle variables such as the width and height of the array are automatically bounded by Halide. The scheduling process can even be done automatically during the compilation by the library, in order to find an optimal schedule on the target platform.

Chapter 2

Preliminaries / Background

2.1 Hero

2.2 Halide Language

2.2.1 Programing model

Halide is a functionnal programming language embedded into C++, designed to write high performance image and array processing code [**?**]. This language uses a functionnal paradigm to describe the the processing pipeline, and disociate the algorithmic code from the it's schedule (how the code will be compiled and run on the system).

Every pipeline is a function (`Halide::Func`) composed of other functions and expressions (`Halide:expr`). These two objects use special variables (`Halide:Vars`) to describe the operation executed on the array. The code snippet 2.1 describe a basic pipeline which compute the distance of each coordinate of a two-dimensional array from on position specified by the vector `(center_x, center_y)`. The creation of the pipeline is straightforward, we have to code the operation we want to execute on every element of the domain (here computing the distance to a given position).

```
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 it into a single stage inlined pipeline or kept as is but allowing only the next stage to execute when the previous one has computed all it's value, or allow it to progress as soon as he has the required data at it's disposal.

After designing the pipeline, we can define it's schedule via the different directives included in Halide. Halide implements basic scheduling primitives that can be combined to create complex schedules. The primitives consists of basic code transformations such as loop unrolling or reordering, loop splitting or fusing variable together, more advanced instructions like parallelization or vectorization are also available. I will explain their behavior more in detail in the section Basic Scheduling Options.

In the example 2.2, we can see how the scheduling works. All instructions are a function of the pipeline object, they take the variables of the pipeline as argument to specify on which part of the pipeline this schedule will affect. During compilation, Halide will generate the code corresponding to the schedule specified by the programmer (if this schedule is valid).

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

Listing 2.2: Simple Pipeline Example

In the example 2.2, we will create as many task as the value x can take. They will then be distributed on the cores of the system. Every task will execute a single loop over the y axis, which will compute ten elements of the output every iterations (as we specified ten as argument of the `unroll` schedule).

The pipeline can be translated or compiled by Halide to be executed directly on the compilation computer or to be used in another application. The function `.realize(x_max, y_max)` is the fastest way to execute the pipeline, and is useful to debug the algorithm or the schedule. As halide was designed to work with different hardware platform, some of them being low power, cross compilation has been simplified to make the process as simple as possible.

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 has tools to debug the pipeline during it's compilation or during execution. First of all, the `print()` can be called any time to print a variable, the `print_when()` which print only when one boolean condition is met.

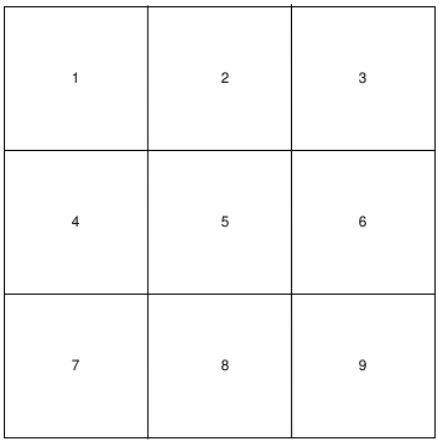
Another useful tool is the `.trace_store()`, this function keeps a trace of every function evaluation in the pipeline. It is possible to get more informations during the compilation of the pipeline by setting the environemental variable `HL_DEBUG_CODEGEN` to 1, this will output information about every stages of the compilation and a pseudo code representation of the pipeline loops. Finally, variables and functions can have a label, which will be used by halide in it's internal representation or when printing the schedule, this function greatly reduce the debugging time of the schedules as Halide gives every variables a different name from the source code.

2.3 Basic Scheduling Options

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

2.3.1 Non Platform Specific Schedule

Default Schedule

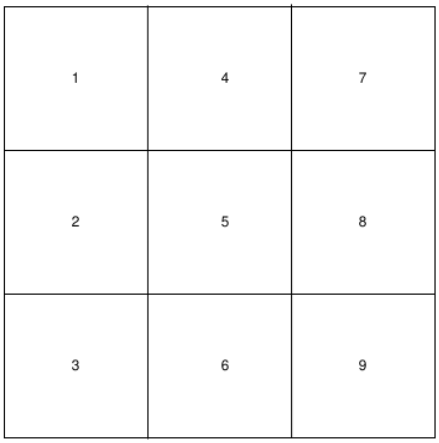



```
for(int y = 0; y < 4; y++){
  for(int x = 0; x < 4; x++){
    do something ...
  }
}
```

Figure 2.1: Base Schedule

If you don't specify any scheduling instructions, Halide will evaluate the pipeline in order. The first variable being the inner loop, and the last one the outer loop. In figure 2.1, we can see the schedule in action, the image is processed in a row major fashion.

Reorder

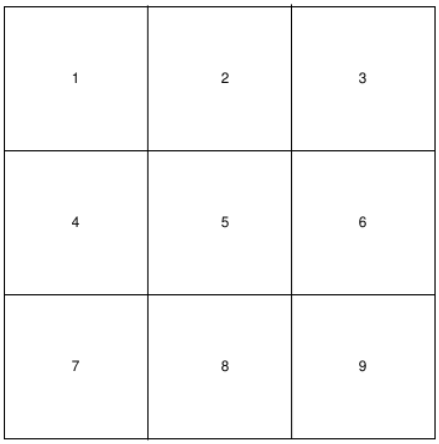



```
  pipeline.reorder(y,x);
for(int x = 0; x < 4; x++){
  for(int y = 0; y < 4; y++){
    do something on
    (x,y)
  }
}
```

Figure 2.2: Schedule: Reorder

The `.reorder` instruction tells Halide how to traverse the domain space, using this instruction we can reorder the loops of the pipeline. In the exemple **??**, we changed the way the array is being processed from row major to column major.

Fuse

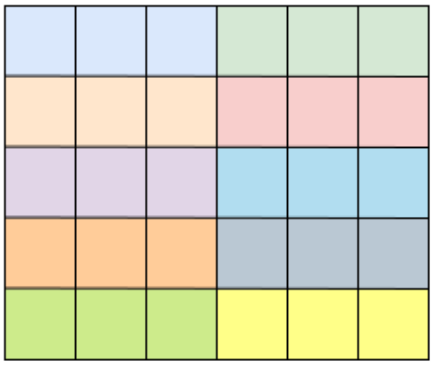



```
  pipeline.fuse(x,y,xy);
for(int xy = 0; xy < 9; xy++){
  do something on
  (xy)
}
```

Figure 2.3: Schedule: Fused

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

Split



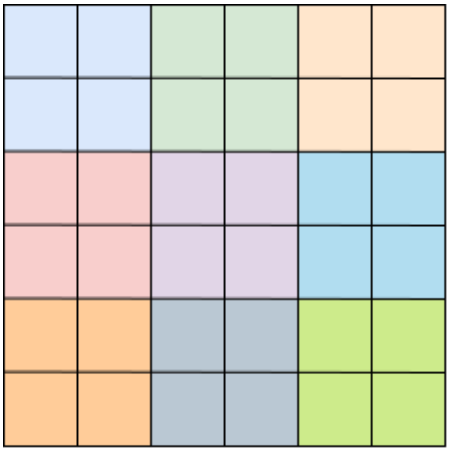
```
pipeline.split(x,x_o,x_i, 3);
for(int y = 0; y < 6; y++){
  for(int x_o = 0; x_o < 3; x_o++){
    for(int x_i = 0; x_i < 4; x_i++){

      do something on
      (x_o * 3 + x_i, y)
    }
  }
}
```

Figure 2.4: Schedule Split

This schedule replaces one loop over a dimension by two loops, an inner loop and an outer loop. This schedule is useful to cut the array in smaller pieces that will be computed in parallel or using Single Instruction Multiple Data (SIMD) instructions.

Tile

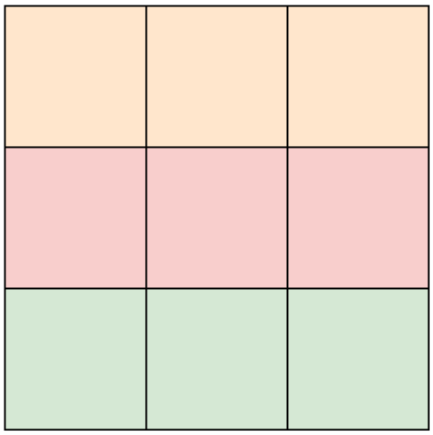


```
pipeline.split(x,y,x_o,y_o,x_i,y_i,2,2);
for(int y_o = 0; y_o < 6; y_o++){
  for(int x_o = 0; x_o < 3; x_o++){
    for(int y_i = 0; y_i < 2; y_i++){
      for(int x_i = 0; x_i < 2; x_i++){
        do something on
        (x_o * 2 + x_i, y_o * 2 + y_i)
      }
    }
  }
}
```

Figure 2.5: Schedule Tile

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

Unroll



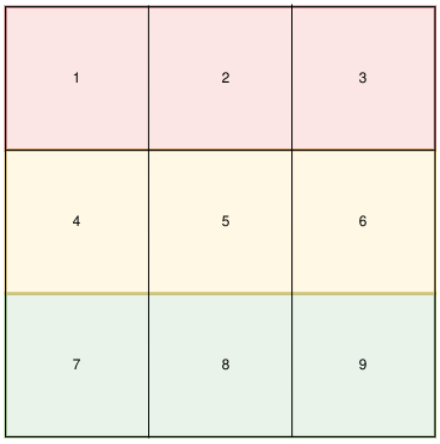
```
pipeline.split(x, x_o,x_i,3);
pipeline.unroll(x_i,3);
for(int y = 0; y < 4; y++){
  do something on (0,y)
  do something on (1,y)
  do something on (2,y)
}
```

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 allows the pipeline to be distributed 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, ans 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 also 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, we can only compile the application to the hardware simulator. The compilation has two phases, during the first one, we compile the Halide application using llvm and run it on the host platform, this application will then generate an risc-V object file and a header. Then we compile the hero application using the already available Makefile, we include the header in the main application and the object file to the sources during the linking command.

2.5 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 riscV code inside the ARM application.

Chapter 3

Design Implementation

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

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

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

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

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

Listing 3.1: Matrix Multiplication Pipeline

3.1 Schedule Implementation

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

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

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

Chapter 4

Results

4.1 Test Setup

I benchmarked two applications on two platforms. I benchmarked the halide port on the hardware simulation for the PULP cluste, and one openMp matrix multiplication application on the developpement platform on a Xilinx ZCU102. For the halide Application I generated random matrices, and for the openMp application I generated them using the same pattern every time. But as one multiplication takes two cycles every time, the execution time doesn't depend on the content of the matrices. The two matrices were stored in the L1 cache, to have the best access time we could, and to compare the code efficiency of both API. 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 resets and output the value of a counter incremented every cycle, as they only take less than twenty cycles to execute, they are useful to get cycles accurate measurements of the execution time of the function. 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 a multiplication or a memory access (which take 2 cycles each), so for a matrix of size n, the number of operations to finish the multiplication is : $2 * n * *3$.

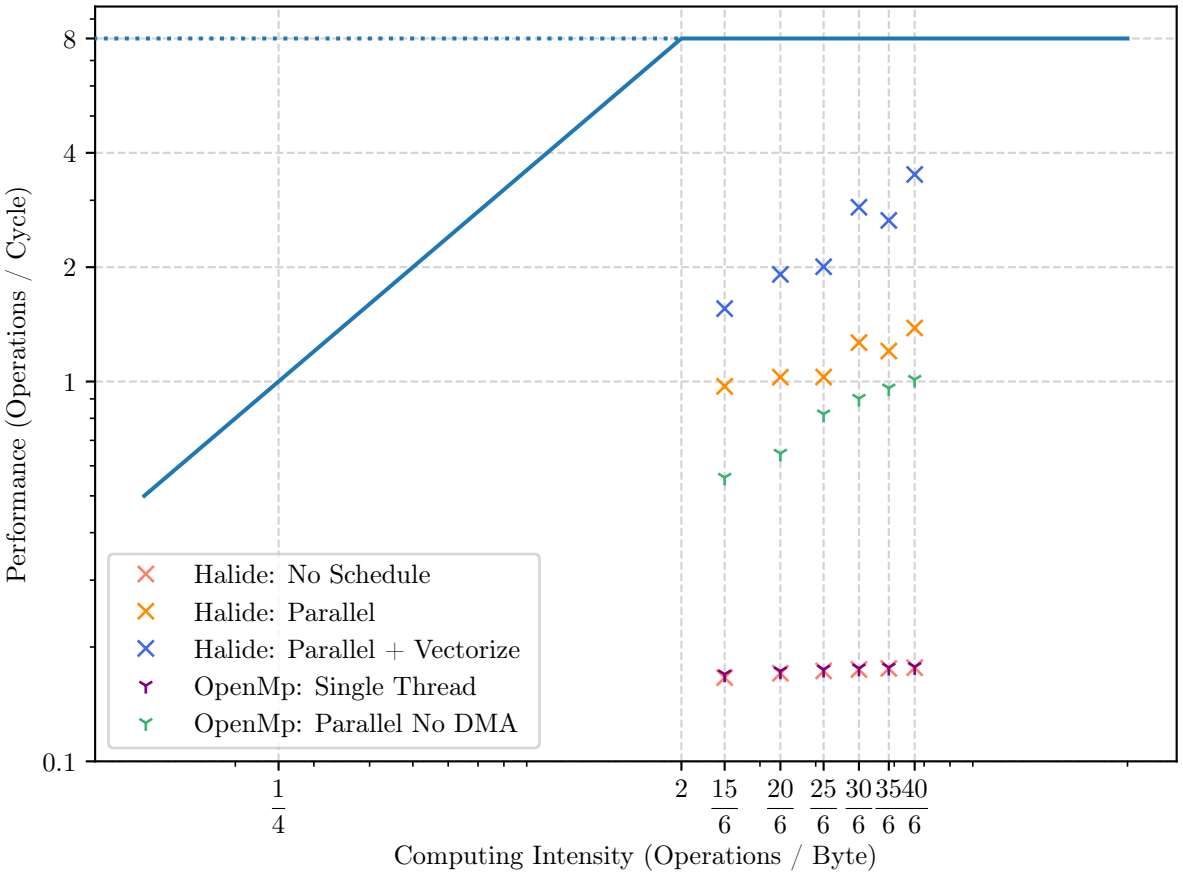


Figure 4.1: A PGF histogram from `matplotlib`.

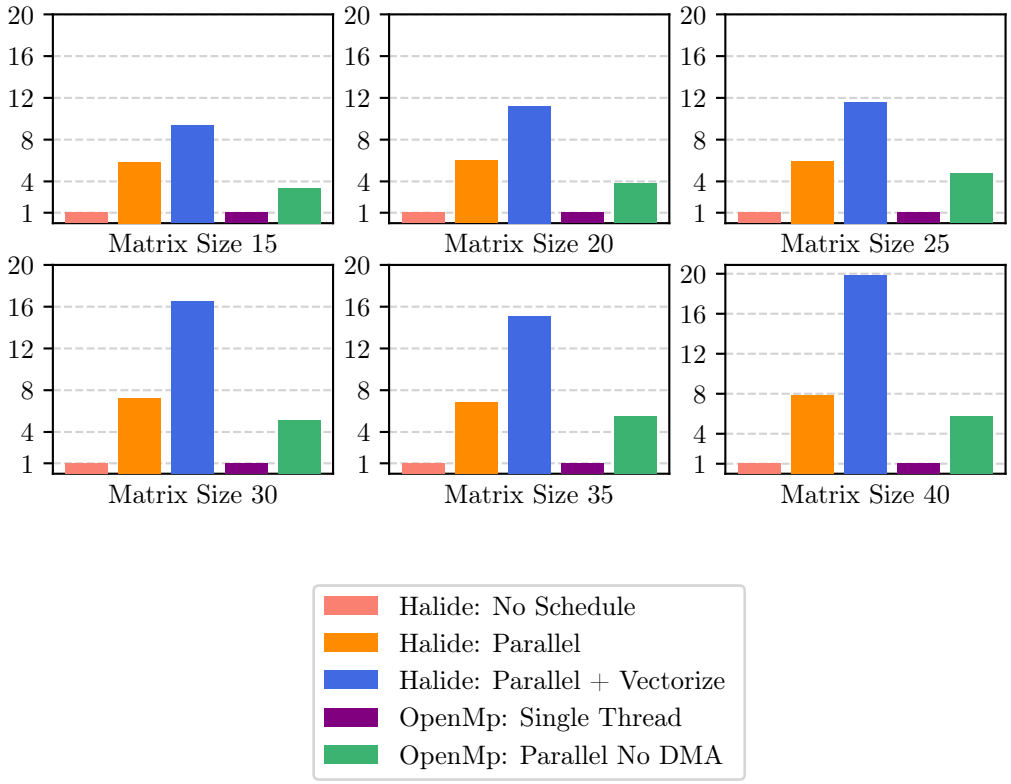


Figure 4.2: A PGF histogram from `matplotlib`.

4.2 Comparaison between OpenMp and Halide on the different platforms

Chapter 5

Conclusion and Future Work

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

5.1 First Section

5.2 Second Section