Bachelor's Thesis Computing Science

Radboud University Nijmegen

FPGA High Performance Computing

Analysing the use and performance of High Level Synthesis toolchains

Author:
Jasmijn Bookelmann
s1052991

First supervisor/assessor:
Prof. Sven-Bodo Scholz

Second assessor:
Dr. Mart Lubbers

August 11, 2024

Abstract

This bachelor thesis is about the evaluation of High Level Synthesis tool chains for Field Programmable Gate Arrays (FPGAs). An FPGA is a device which can be reprogrammed with developed hardware circuits. HLS is used to generate this circuit: it is a tool which translates a high level language into a circuit design.

We evaluate how well FPGA HLS tool chains can be used for High Performance Computing, and how much hand optimisation and low-level hardware knowledge about the FPGA board is required. In order to accomplish this, we conduct two case studies using Vitis HLS. A vector addition and a 2D 5-point stencil computation. For both of these case studies we start with a naive version of the implementation, and then iteratively attempt to improve the performance. Then we compare the performance of final implementation against a parallel CPU and GPU version, and the energy consumption of the FPGA against the GPU.

Our results show that there is more than a 200-fold difference between a non optimised version and a hand optimised version. In addition, in order to make these improvements, domain knowledge of hardware design and the card structure is required. Our comparisons against a CPU and GPU show that the CPU and GPU had better performance than the FPGA, however the energy consumption of the FPGA is better than that of the GPU.

Contents

1 Introduction **3**

2 Background **5**
2.1 FPGAs 5
2.2 High Level Synthesis 6
2.3 The accelerator card 7
2.4 Using Vitis 8
2.4.1 Host code 8
2.4.2 Kernel code 9
2.4.3 Configuration files 10
2.4.4 AXI 10
2.5 OpenCL 11
2.6 Performance Metrics 11
2.6.1 Performance 12
2.6.2 Performance per Watt 12

3 Research Approach **13**
3.1 General approach 13
3.2 Technical approach 14
3.2.1 Running code 14
3.2.2 Performance measurement 14
3.2.3 Energy measurement 18

4 Vector addition **22**
4.1 Naive solution 22
4.2 AXI port optimisation 23
4.2.1 Design 23
4.2.2 Implementation 25
4.2.3 Results 29
4.3 Using multiple DDR cards 29
4.3.1 Design 29
4.3.2 Implementation 30
4.3.3 Results 30

4.4 Multiple FPGA kernels 31
4.5 Comparison to GPU and CPU 33
4.5.1 Implementation 33
4.5.2 Performance 34
4.5.3 Energy consumption 35

5 5-point stencil computation 36
5.1 Naive implementation 36
5.2 Getting the code to work 38
5.2.1 Decreasing the amount of accesses to the coefficients port 38
5.2.2 Adding a cache to the in port 39
5.3 Improvements by improving dataflow 40
5.3.1 Initial Design 40
5.3.2 Improvements by widening the dataflow width 41
5.3.3 Implementation 43
5.3.4 Results 46
5.4 Comparison to GPU and CPU 47
5.4.1 Implementation of CPU and GPU 47
5.4.2 Performance 48
5.4.3 Energy consumption 49

6 Related Work 50

7 Conclusions 52
7.1 Practical observation 53
7.1.1 Technical installation and use process 53
7.1.2 Compilation times 53
7.2 Future work 54

A Appendix 59
A.1 Example of host code 59
A.2 Host code with multiple kernels 61
A.3 Stencil code 63

Chapter 1

Introduction

In the 1977 Turing award lecture, the Von Neumann bottleneck or "memory wall problem" is established: Every time a typical von Neumann computer executes a single step, it has to fetch an instruction from a store, fetch the data, perform this single instruction, and store the result.

There are two important limitations due to the memory wall problem: Data movement is greatly limited by memory bandwidth. In addition, there is high power consumption, as the majority of power consumption is caused by moving data between computing and off-chip memory units [26]. Both of these limitations have only increased with current development: CPUs can process data even faster compared to their memory bandwidth because of pipelining and multithreading. And power consumption has become a bottleneck in and of itself since the breakdown of Dennart scaling, meaning that smaller chips consume more energy.

We can avoid these limitations by using Field Programmable Gate Arrays (FPGAs). FPGAs are devices on which developers can program custom circuits. This means that it is possible to program custom data paths, unlike CPUs and GPUs, which both rely on a fixed circuit. FPGAs don't have to fetch instructions, they are built into the physical implementation itself and data "flows through this implementation" [20]. This means that FPGAs consume relatively little power, as there is no overhead from fetching instructions. Due to these benefits FPGAs have slowly been adopted as hardware accelerators over the past years. [21].

However, there is a disadvantage when employing FPGAs: they are difficult to program. While developers are able to make use of custom circuits, this also means they have to define custom-made circuits.

Most FPGAs are programmed using hardware description languages (HDLs). These languages define the individual wires and gates of a circuit. Because of this, a lot of development time and hardware expertise is required when programming using these languages.

For the past decades there has been research in order to solve this, through the development of High Level Synthesis tools. These tools take a higher level description, such as a high level programming language, and then compile this to a circuit design. [7].

This thesis investigates the effectiveness of one instance of these HLS tool chains. We do this by implementing two applications on the FPGA. First a vector addition, and secondly a 5-point 2D stencil computation. The vector addition has low complexity. This allows us to focus on proper configuration and optimising the throughput of the FPGA. The stencil computation has higher complexity and memory access patterns. This enables us to focus on the implementation of the problem itself.

We initially write a 'naive' version of both these programs, and see how well these perform. Then we iteratively try to improve the performance, looking at how much performance increase we get and how much domain knowledge is required for these optimisations. Finally we compare the performance of the resulting implementation to a CPU and GPU implementation, and the energy usage to a GPU implementation.

While there is a variety of devices and tools available, we use the available AMD Alveo U200 accelerator card, which we program using Vitis, as this is the environment AMD provides for the card.

Chapter 2

Background

2.1 FPGAs

GPU and CPUs both have fixed hardware: Their circuitry is fixed upon production and cannot be changed. The way CPUs and GPUs execute tasks is by reading instructions. Instructions tells the CPU what to do, for example to add two items together, or to move a piece of data from one location to another.

An FPGA is different, it does not have fixed hardware. Instead it acts as a set of gates, registers and wires which can be connected in any way by programming it. This allows developers to design custom circuits to execute specific tasks.

FPGAs can be considered a more general form of a CPU or GPU. For example, it is theoretically possible to program a CPU on an FPGA. However, FPGAs are not just a better version of CPUs or GPUs. In order to achieve the flexibility of dynamic hardware, constraints are introduced on the possible circuits which can be programmed onto an FPGA.

Resource limits An FPGA has a limited amount of ”space”. Broadly speaking, FPGA space is measured in lookup tables (LUTs), block RAM (BRAM) and digital signal processing blocks (DSPs). LUTs are used for the logic operations, BRAMS are used for storing data and DSPs are used for arithmetic operations.

There is a finite amount of each of these on the FPGA, limiting the circuit design. For example if you want to allocate memory on the FPGA to store a very large array, this would be limited by the BRAM space.

Clock speed A clock is a device which generates pulses for the hardware circuit, for every pulse the circuit goes into it’s next state, so it ”updates”. The frequency of the pulses is called the clock frequency. The higher the clock frequency, the quicker the state of the circuit updates, so the faster a program executes. Clock speed cannot be arbitrarily increased, as every time a pulse is sent, the circuit needs to first be stable again. If the clock sends pulses too quickly, the circuit will not have time to stabilize, and we get unpredictable, non-deterministic behavior.

While stabilising, the pulse "travels" through the circuit, and once it has reached all components of the circuit, it is stable again. The longest path this pulse needs to take is called the critical path. The time of the critical path determines the maximum clock speed.

For this thesis, the most relevant parts that influence the critical path are: Simply the length of a path, if we place components very far away from each other the critical path will be low. In addition, doing a series of computation right after one another without any storage in between also lengthens the critical path significantly.

Compare to a CPU or GPU an FPGA has a lower clock speed: where GPU or CPU clock speeds are usually in the 1-10 GHz range, FPGAs usually operate in the 100-300 MHz range. This means that while FPGAs tend do more per clock cycle, there are less clock cycles per second compared to a CPU or GPU.

2.2 High Level Synthesis

In order to program an FPGA, we need a circuit design.

Most, if not all, modern FPGAs take hardware description languages as descriptor. These languages describe a hardware circuit, so how e.g. gates, wires and registers are all connected.

High Level Synthesis tools take a higher level description, such as a C based language, and then synthesize this to become a circuit design which can be used on the FPGA.

This allows developers to design circuits with more efficiency, and requiring them to have less domain knowledge. Developers don't need to be able to create hardware designs, understand the specifics of the FPGA board they're using or understand hardware level communication protocols in order to create a circuit using HLS. Instead they can program an algorithmic implementation which is then compiled to a circuit.

Usually an FPGA is not used by itself, but is part of a card containing various connections to memory and devices. Configuring these connections is called system level integration. HLS itself is not responsible for connecting these components to the FPGA. Instead this is done by a different step in the tool chain called linking. [8].

This is why in our thesis we research "tool chains" instead of "tools" or "languages", as we are also making use of and evaluating the linking step.

2.3 The accelerator card

The FPGA accelerator card we used is the Alveo U200. This card does not only contain an FPGA, but also memory and other components. Similar to a GPU card, we physically connect this card to the host computer.

You can see in the following infographic roughly what the card looks like: [1]

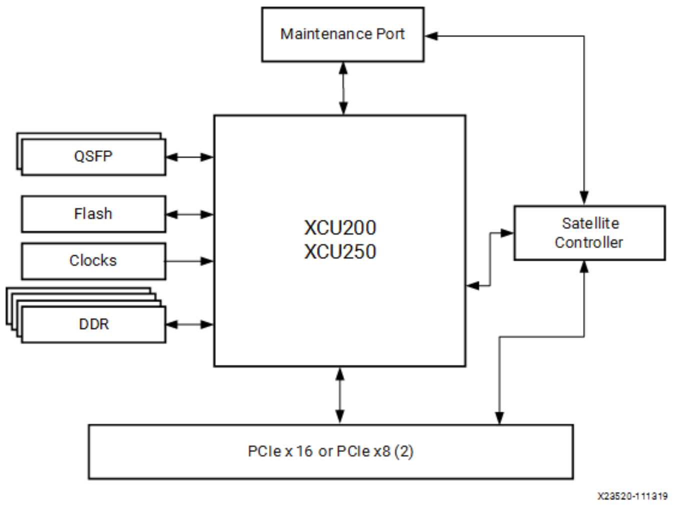


Figure 2.1: Alveo U200 diagram

The relevant connected components are:

- The PCIe interface, which is a channel used to receive or transfer data to or from other components. In our case we use it to communicate with the CPU.
- The 4 DDR cards, these cards each contain 16 GB of data, so 64 GB total. These can be directly accessed by the FPGA. The maximum bandwidth of all cards total is $77\,\mathrm{GB\,s^{-1}}$. This is based on the fact that each card has one access port, and each of these access ports can theoretically read or write 512 bits per clock cycle at a maximum speed of 300 MHz.
- The FPGA itself (XCU200). The FPGA is divided into 4 separate Super Logic Regions (SLR). The 4 DDR cards and other components are divided over these SLRs. SLR region crossings can be resource intensive, which can have an impact on whether a circuit can be programmed onto the FPGA or achieve a high clock frequency.

2.4 Using Vitis

As the Alveo chip is from AMD, we use their toolchain: Vitis. In this section, we only discuss the necessary information about Vitis. The complete documentation can be found at [22].

Vitis contains a multitude of sub-components, such as a programming environment, debugger and various other tools. Vitis also has a HLS language: "Vitis HLS". This language translates C++ into HDL code. Then the user can link the HDL kernel to the device to create a kernel binary. This binary can be programmed and ran on the FPGA. Vitis generates three different reports based on these steps: The compilation, linking and runtime report.

In order to use the FPGA, we need to write two applications:

- The host application: This application runs on the CPU and is responsible for loading and calling the FPGA kernel.
- The kernel application: This application is compiled to a HDL, linked and then loaded onto the FPGA

2.4.1 Host code

The host code is responsible for programming the FPGA with a kernel, loading data onto the FPGA and receiving data from the FPGA.

The host code can be created using two different tools: the xrt library and OpenCL. xrt is a library from AMD responsible for communication with the FPGA. The OpenCL implementation wraps xrt. We use OpenCL as this gives us access to OpenCL's framework, which allows for easier profiling and parallel execution of tasks.

The host code executes the following steps:

1. Find the FPGA accelerator card
2. Load the kernel onto the card
3. Transfer the input data to the global memory of the card (usually DDR memory)
4. Run the kernel
5. Transfer the data back

An example of host code can be found in A.1.

It's important to note that, unlike GPUs, you cannot compile the kernel code at runtime. This is because creating the kernel binary tends to take quite a long time, usually 3 - 9+ hours depending on the complexity.

2.4.2 Kernel code

Kernel code is written in either C++ or OpenCL, and then compiled to a HDL language by Vitis HLS. We use C++, as there is no documentation of OpenCL, and all examples using OpenCL kernels have been removed as of Vitis version 2023.2. [24].

The kernel code structure is very similar to a normal C++ program and it is possible to use C++ features such as classes. However, most libraries except for a select few should be avoided because they are not written with HLS compilation in mind, meaning they are either extremely inefficient or won't work at all.

The kernel code base structure looks as follows:

```
extern "C" {
    void fpga_kernel(kernel arguments) {
        ...
    }
}
```

Where `fpga_kernel` is the top function of the kernel. The kernel arguments represent data transferred from the host to the kernel. Throughout this thesis, we also refer to these arguments as ”ports”, considering these create ports from the FPGA to the host.

Kernel code is compiled in two different steps: First you synthesize the code to a HDL kernel, after that you link the kernel to the specific device to create a kernel binary.

HLS Pragmas

Vitis has various pragmas which can be used to communicate to the compiler how code should be synthesized. They can be applied using:

```
    # pragma HLS [pragma name] [pragma arguments]
```

Pragmas can control the following aspects of the code:

- Mapping of variables to hardware: For example arrays can be mapped to individual registers, BRAM or FIFOs. This has impact on the access patterns and resource usage.
- Kernel optimisation, for example whether functions can be executed at the same time or execution order can be optimised. Also whether loops can ”flattened” and every single item executed at the same time.
- Interface configuration: How ports are configured, for example which protocol they use and how this protocol is configured.

You can find a detailed description of all Vitis HLS pragmas at [9].

The Vitis HLS library

Vitis HLS provides its own libraries to help with program design.

The two libraries relevant for this thesis are the HLS Stream Library and the HLS Vector Library.

The HLS stream library provides the `hls::stream` class, which represents a FIFO queue. Meaning that we can write to the back of the queue using the `write` function, and read from the front using the `read` function. Streams are generic and can be used to contain any type of data with a fixed size. The maximum items the stream can contain is fixed at compile time.

The HLS vector library provides the `hls::vector` class, which represents a vector of elements. The size of this vector is fixed at compile time. It is possible to create streams of vectors, or use vectors as top-level arguments.

2.4.3 Configuration files

We have 2 configuration files which control the compilation and linking of the kernel code:

- `config.cfg`: This file configures the compilation of the hardware kernel. For example properties such as how arrays should be partitioned by default.

- `link.cfg`: This files configures the linking of the hardware kernel to the host. It mostly configures the connection between host and kernel. Such as which memory parts ports should be mapped to.

2.4.4 AXI

AXI is the protocol used to communicate between the FPGA and other components, including the host and DDR memory. When kernel code is synthesized, all the top level arguments are implemented as AXI ports. The AXI ports are then connected to a specific memory component during linking.

In this section, we do not explain the exact details of the AXI protocol, instead we show a rough overview of how data transfer works.

A data transfer to the FPGA works as follows: The FPGA kernel requests data at a certain address. The kernel then waits for a fixed amount of cycles before assuming the data has arrived. In the meantime, the external memory receives this request and replies back with the data.

The FPGA has a maximum amount of outstanding requests, meaning requests for which there is no reply yet.

Instead of requesting single items of data, it is also possible to request multiple sequential items of data. This is called AXI burst. Bursts are realised in two ways: First the memory device just sends more items back sequentially based on a single request. Second, up to 512 bits of data can be transferred every clock cycle. If we have items of 32 bits, this means 16 items can be transferred every cycle. This is called port widening, as it requires the ports to be a width of 512 bits.

2.5 OpenCL

OpenCL is a framework for parallel computing on heterogeneous systems. Heterogeneous systems are systems which contain multiple different types of devices. Such as a CPU and GPU, or in our case, a CPU and an FPGA.

OpenCL code is usually executed on the CPU: The host device. OpenCL provides a defined interface used to send commands to the other device, it is up to the vendor of the device to actually implement these functions. This means that while the functions are the same, the actual implementation of these functions is different.

When running an OpenCL application, the steps are as follows:

- Find the device
- Load the kernel onto the device
- Create a command queue
- Add commands to this queue, these will be executed on the device.
- Wait until the queue is finished.

In order to execute something on the FPGA or GPU the specific commands are similar to what we described in 2.4.1.

How these commands are executed is different for the two devices. When running the kernel on a GPU, usually the `enqueueNDRange` command is used to divide work groups among many threads and execute these. For FPGAs this is different as there is usually only a single "thread" (kernel). So instead `enqueueTask` is used to enqueue a single task. It is possible to program more kernels onto the FPGA so kernels can run in parallel. When doing this, the first available kernel is used when using `enqueueTask`. However arguments still need to be set manually, and compared to a GPU there are still a lot less "threads" (kernels). In our research we use up to 4 kernels.

More information about using OpenCL for Vitis can be found at [18].

2.6 Performance Metrics

In order to compare the performance and energy usage of different devices, we need to have common metrics.

2.6.1 Performance

We measure performance in FLOPS. This stands for Floating Point Operations per second. A floating point operation is a single operation on a 32-bit floating point variable (in C++ this is a `float`). For example an addition, subtraction, multiplication or division.

In this thesis we refer to the multiple of Floating Point Operation as "FLOPs". And the multiple of Floating Point Operations per second as "FLOPS".

2.6.2 Performance per Watt

Performance per watt is a metric used to measure energy efficiency.

Performance per watt is defined as follows:

$$\frac{\text{Performance}}{P} = \frac{\text{FLOPS}}{\text{watt}}$$

In order to clearly identify what this metric tells us, we can rewrite this formula:

$$\frac{\text{FLOPS}}{\text{watt}} = \frac{\text{FLOPs}/s}{\text{watt}/s} = \frac{\text{FLOPs}}{\text{Joule}}$$

So performance per watt tells us how many floating point operations we can execute using a single joule of energy.

This measurement can be important in various scenarios. For example, in data centers, where energy usage has been steadily increasing over the years and is considered a cost factor [11], it can be valuable to have a device which consumes less power for doing the same amount of operations.

Chapter 3

Research Approach

3.1 General approach

In order to answer my research question, we implement two applications:

1. A vector addition
2. A 5-point 2D stencil computation

Vector Addition Vector addition means the sum of two vectors. Given an input of two vectors, the output should be the sum of these two. For example for inputs `{1.2, 2.4, 3.1}` and `{4.6, 5.3, 6.7}`, we should get the output `{5.8, 7.7, 9.8}`.

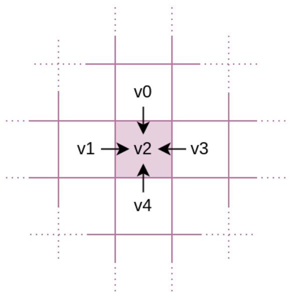
The vector addition problem is chosen because it is a single computation over each value in two lists. This makes the actual computation arbitrary, and allows us to focus on increasing the bandwidth of the FPGA: How quick can we get data into the FPGA, and how quick can we get data back.

5-Point Stencil Computation A stencil computation is a computation which processes values according to a predefined pattern. A 5-point 2D stencil computation is a specific kind of stencil computation: This is a computation which computes the new value of a point based on its direct neighbours in a 2D array.

Given two inputs: A 2-dimensional array and a list of 5 coefficients, the program outputs a new 2-dimensional array. The output value of a point as follows:

$$\begin{aligned} v_{x,y} = {} & c_0 \cdot v_{x-1,y-1} + c_1 \cdot v_{x,y-1} + c_2 \cdot v_{x+1,y-1} \\ & + c_3 \cdot v_{x-1,y} \end{aligned}$$

Output values on the border, which don't have all 4 neighbours, are kept the same as the input value.



The stencil computation is a more complex computation, which requires more complex memory access patterns. This allows us to focus on the actual computation itself.

Approach For both problems we first start out with a "naive" version. This version is the simplest implementation of the specified problem. It is not aware of any hardware specific optimisation.

We then look into whether this program works when synthesized to FPGA code, and if it does, how well it performs. Then we iteratively try to identify issues and bottlenecks and improve the performance.

We also note how much actual code structure change is necessary for those optimisations and how much domain knowledge is required.

Lastly, for each application we create a CPU and GPU implementation. We compare the optimised FPGA application with the best performance to these implementations. Here we look separately at the performance of the entire application and that of the kernel. The application time includes the transfer times, and the kernel time excludes these.

In addition, we compare the FPGA application to the GPU in terms of energy consumption. Here we look at total energy used and performance per watt.

3.2 Technical approach

3.2.1 Running code

We run code on an Ubuntu 20.02 server through slurm. This server has both the Alveo card and the GPU installed. The CPU has 16 cores and 32 threads. The GPU is an NVIDIA A30.

3.2.2 Performance measurement

Performance, measured in Floating Point Operations per Second can be calculated as follows:

$$\text{FLOPS} = \frac{\text{FLOPs}}{\text{s}}$$

The number of FLOPs can be calculated using the amount of items we have as input.

$$\text{FLOPs} = \text{item amount} \cdot \text{FLOPs per item}$$

”FLOPs per item” is a fixed number dependent on the case. For the vector addition, this is 1, as we only have to do a single addition for each item in the length. For the stencil computation this is 9, because for every item we need to do 5 multiplications and 4 additions. For the items on the edge, this is not the case. However we do not take this into account as for large data sizes, the amount of items on the edge is negligible.

We do need to measure the time it takes to run the application.

There are for two different purposes for measurement: Measurement for optimising the FPGA application, and measurement in order to compare the final performance to CPU and GPU. For these we have two different approaches.

Measuring the FPGA when optimising

The goal when measuring when optimising is to see how an application with an optimisation compares to an earlier version.

We use OpenCL queue profiling to measure allocation and kernel times individually. We also use the C++ chrono library to measure the time by measuring the wall-clock time before queueing anything on the queue and after finishing the queue. The individual times summed should be equal to the total time, which holds as shown in the following figure.

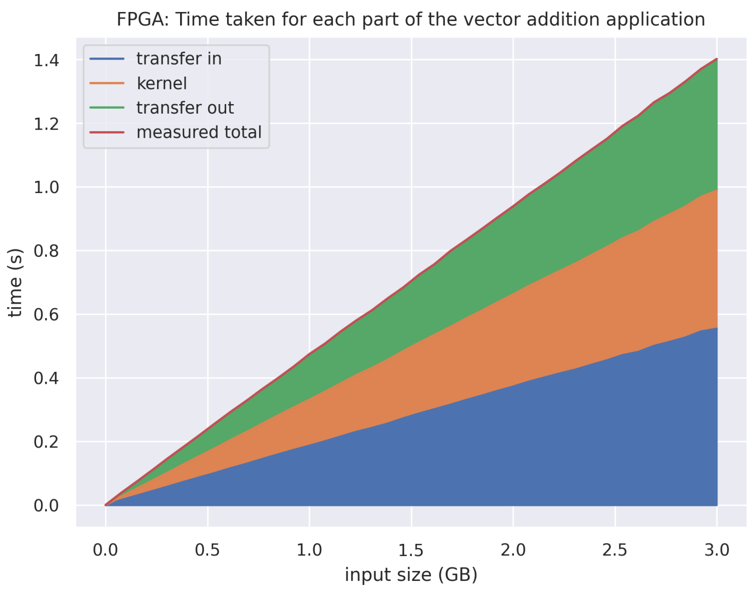


Figure 3.1: Kernel time vs total time for FPGA

We measure the data by running the application for a fixed data size. This is because the kernel is stable and deterministic, meaning variable data sizes do not result in different performances of the kernel itself, only changing the relative overhead from calling the kernel.

We choose a size of 1 GB. As the maximum buffer size for a single input is 4GB [5], and for our optimised vector addition applications we find that after 1 GB of input the application times are almost constant.

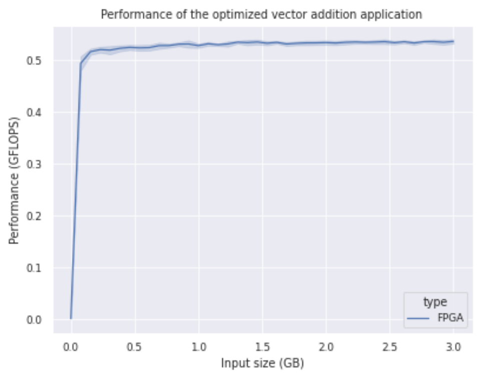


Figure 3.2: For all sizes larger than 1 GB the application performance is almost the same as the performance found at 1 GB, with a maximum deviation of 3%

Another advantage of the stableness of the kernel is that we don't need to measure the kernel performance multiple times. So in cases where we are only looking at how to improve the kernel runtime, we only run our tests once.

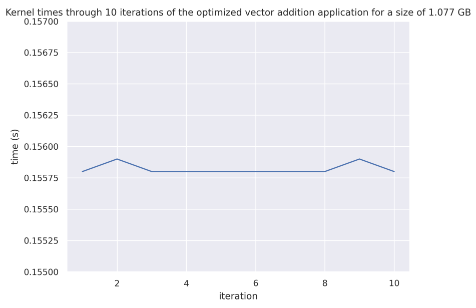


Figure 3.3: Kernel times stay within a range of 0.1558 and 0.1559 for all 10 measurements

If this is not the case, and other elements also have impact, such as allocation time, we run the tests 10 times and take the average.

While this is the default method for measuring when improving, if we believe that measuring more extensively gives valuable insights we will do so.

Measuring for comparison

As the other devices are a lot less stable than the FPGA kernel, and transfer times are also included, we measure the data for 40 different sizes, for 10 iterations each. These parameters are the result of balancing the highest possible amount of sizes and repeated measurements with a reasonable measurement time of under half an hour.

FPGA time measurement We measure time the same way as described in 3.2.2.

GPU time measurement As we are also using OpenCL for the GPU, we can attempt to measure time similar to how we measure the FPGA. However when doing this there is a disconnect between the total time and the sum of the individual event times:

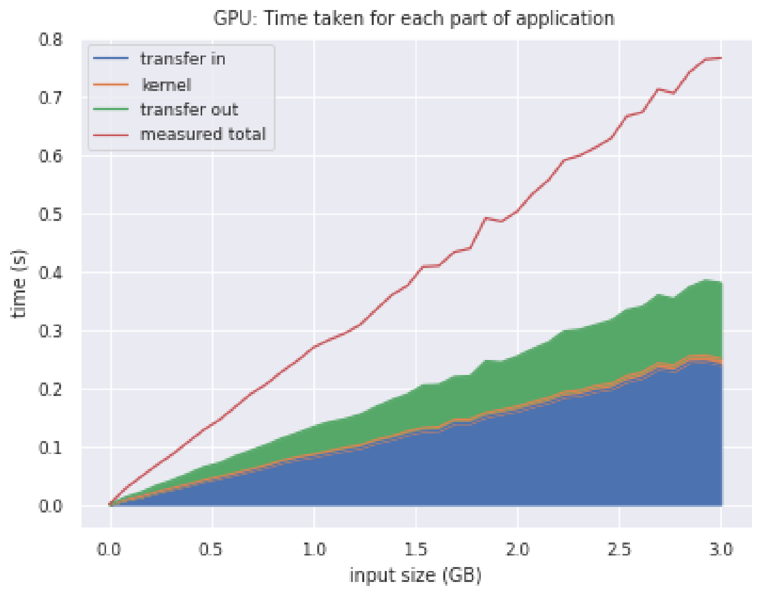


Figure 3.4: Sum of individual times vs total time for the GPU vector addition application when using the OpenCL profiling times

This is likely because of a different OpenCL implementation by NVIDIA of the event profiling of the transfer calls. So instead, we use wall-clock time for each separate event: the transfer in, running the kernel and the transfer out. We do this by finishing the queue to make sure the last command has finished executing, measuring the time and then queuing the next command. The disadvantage to this is that there is overhead due to having the wait for the queue to finish before being able to execute the next command.

CPU time measurement For the CPU, we also use the chrono library to measure time, similar to 3.2.2. However, we do not need to transfer any data from host to device, as the host is the CPU. Because of this we set the transfer times to 0. This also means that the application performance and the kernel performance is the same for the CPU.

3.2.3 Energy measurement

In order to compare FPGA energy usage with GPU energy usage, we need to measure the energy.

We measure the energy by executing a bash script which periodically fetches the energy usage and prints the time. The host application also prints the time right before running the kernel, and the time right after.

Using python, we process the resulting data: We cross-reference the times to select an accurate range from the measurement results and calculate the resulting energy usage.

Using bash

We start this bash script when the host application starts, and stop it when the host application finishes:

```
measure-power-loop 0.001 >> "$OUTPUT_FILE" &
// run the application
...
sleep 2
kill $!
```

It can also be seen that we sleep for two seconds after the execution of the program. This is to make sure all delayed power usage is captured.

In order to measure the FPGA power, we use the utility command ”xbutil” which is part of the Vitis tool chain. For the GPU we use the ”nvidia-smi” command.

```
# measure the current power of the FPGA:
OUTPUT="$(xbutil examine --device 0000:e2:00.1 --report
    electrical)"
echo "$OUTPUT" \
  | grep -Eo '^\s+Power\s+:\s+\S+' \
  | awk '{print $NF}'

# measure the current power of the GPU:
nvidia-smi --query-gpu=power.draw --format=csv,noheader,nounits
```

Pauses

We are running the application multiple times in order to get our measurements. In order to clearly separate the runs from each other, we artificially pause for 1 second between iterations.

Selection of measurements

When cross referencing the data we get the following coverage:

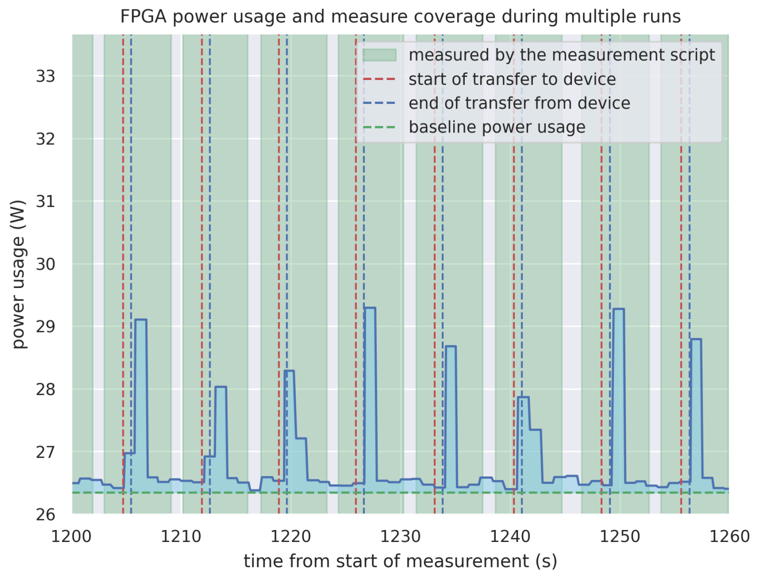


Figure 3.5: FPGA energy usage and measurement coverage throughout runs

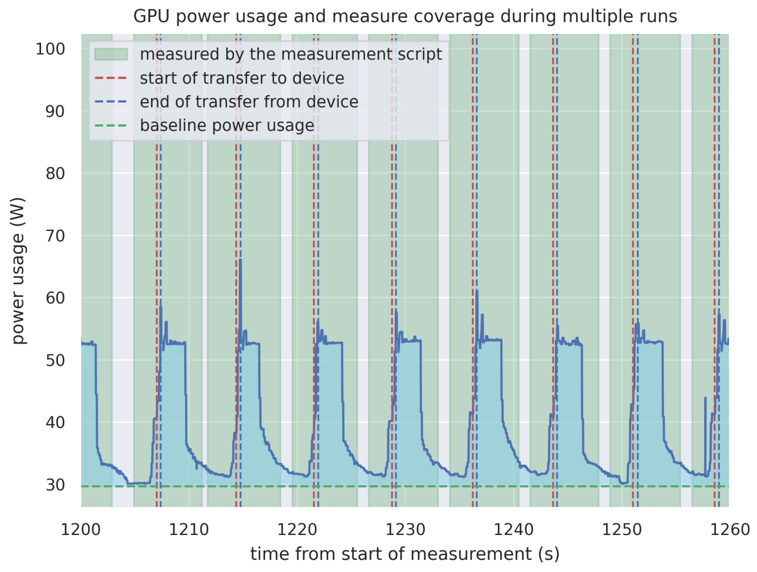


Figure 3.6: GPU energy usage and measurement coverage throughout runs

It can be seen on the figure that both devices still appear to draw energy after they have finished. We need to also take this into account.

In addition, for the GPU, before the running of the kernel, there is energy usage before the transfer in. This is due to the programming of the kernel onto the GPU, which is done for every run. This is not the case for the FPGA, because once the FPGA is programmed with a certain kernel, if we attempt program it again with the exact same kernel, instead of programming the kernel again it will just keep the original kernel.

We are not going to take kernel loading energy usage and times into account, as this would require us to program the FPGA with another different kernel between every measurement.

So our final measurement slot is from the start of transfer in, to the end of the coverage of the measurement script:

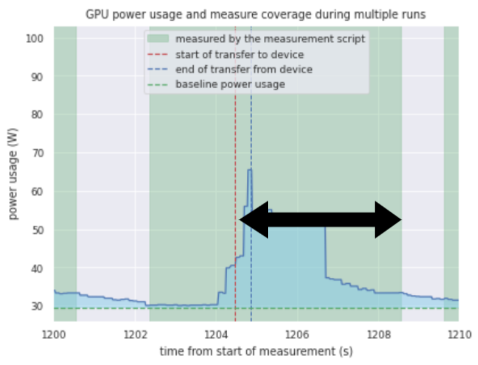


Figure 3.7: The range used for energy measurements

This way we avoid the energy usage caused by the programming of the kernel while at the same time covering the full energy usage of the device as result of its use.

Calculating the energy usage

We measure the power. Based on these measurements, we can calculate the energy usage of one run as follows:

$$E = P \cdot t = (P_{avg} - P_{base}) \cdot (t_1 - t_0)$$

Where

- P_{avg} is the average power usage during the run in watts
- P_{base} is the base power usage in watts, this is the watt value when the device is not doing anything, for this we use the lowest measured value throughout all runs, usually this is the value completely at the start, when the device hasn't done anything yet. For this we do use all values we measured.
- t_0 is the time of the first measurement within our range in seconds
- t_1 the time of the last measurement in seconds

Calculating performance per watt We calculate the performance per watt as follows:

$$\text{Performance per watt} = \frac{\text{Performance}}{P}$$

Where:

- Performance is the speed of computation in FLOP per second.
- P is the power usage. For each run, this is the maximum value measured within the range. We cannot use the average power measured, as this also includes times when the kernel is not running, and as for both the FPGA and GPU measurements the "peaks" of the kernel consumption are flat, we can use the maximum.

Accuracy

Measurement frequency The frequency of the FPGA measurements is about 1 measurement per second. This means that for application times shorter than this interval, there is a possibility that there is no measurement while the application is running. This leads to inaccurate results.

During our analysis we keep in mind that the FPGA measurements for small data sizes are inaccurate, and look at larger data sizes. We also measure for 10 iterations per data size, and take the average to get a more accurate result.

In contrast, the GPU frequency is 0.1 seconds, which is a 10 times higher frequency. This means that the GPU measurements is more accurate.

Baseline discrepancy The baseline power for the GPU is higher than that of the FPGA. The GPU baseline is equal to 29.61 and that of the FPGA to 28.13. This means that the GPU does consume more power while not doing anything which is not accounted for in the results as we subtract the baseline from the current power.

Chapter 4

Vector addition

As mentioned before in 3.1, vector addition is defined as the sum of two vectors.

We take two input vectors, called `in1` and `in2` in the code, then we sum these two input vectors and store the result in a third vector, which we call `out`. All vectors are the same size.

4.1 Naive solution

The naive implementation of the vector addition kernel is shown below. The kernel code is implemented as a single function with the input and output defined as parameters. `in1`, `in2` and `out` are the input and output vectors. `size` is the size of all three vectors. The kernel code iterates over all indices, adding the input vectors together and writing this to the output vector.

```
typedef float data_t;
typedef uint32_t size_int;

extern "C" {
    void fpga_kernel(
        data_t* in1,
        data_t* in2,
        data_t* out,
        size_int size
) {
      for (int i = 0; i < size; i++) {
          out[i] = in1[i] + in2[i] ;
      }
    }
}
```

Listing 4.1: Naive vector addition kernel code

The host code can be found in the appendix as A.1. The general approach of the host code is similar what is described in 2.4.1, so the code first finds the device, transfers `in1` and `in2` to the card, executes the kernel and transfers

`out` back. The host code is the exact same for all implementations in this chapter unless explicitly described otherwise.
We apply HLS to this kernel code to create a kernel binary, and then measure the resulting application as described in the technical approach section.
Note that the input and output vectors we put in the parameters of the source code are compiled to become kernel ports. So the kernel speed per port is the read/write speed of a single port, of which the current kernel has 3. The size parameter is not a port as it's only a single variable.
We find the following performance:

	size (GB)	time (s)	speed (GB s^{-1})	performance (GFLOPS)
transfer in	2.00	2.49×10^{-1}	8.02	-
kernel (per port)	1.00	2.50×10^{1}	4.00×10^{-2}	1.00×10^{-1}
transfer out	1.00	9.37×10^{-2}	1.07×10^{1}	-
total	1.00	2.54×10^{1}	3.94×10^{-2}	9.86×10^{-2}

We can observe that most time is spent in the kernel itself, on the computation. This is surprising because there is very little computation to be done.
Upon investigation by looking at the Vitis runtime report we find the cause of this: the transfer speed between the kernel and external memory. More specifically, the configuration of the AXI connections between the kernel and the DDR. It is stated that the kernel read utilisation is 0.212% and the kernel write utilisation is 0.416% in relation to the utilisation using ideal port configuration. In addition, the report also states that the kernel has an external memory stall of 93%, so the kernel spends 93% of the time not doing anything and waiting for data to arrive from the DDR.

4.2 AXI port optimisation

4.2.1 Design

In order to explain the cause of the low transfer speed, we need to look at the AXI configuration. In the compilation report it is stated how the AXI ports are currently configured: AXI burst is not enabled, and with this, the port width is set to be equal to the data type used: 32 bits.
This means that that sequentially, for every one of the 3 vectors, the kernel is sending a few requests for 32 bit items. Then the kernel waits for a fixed amount of cycles until the data has been received or acknowledged. Only after this has happened, does the kernel send another request.
This explains the low transfer speed and high stall percentage found in the results. In order to improve the bandwidth we can enable AXI bursts. By requesting more data at once, stall time can be minimised.

Sequential bursts

If we just enable sequential bursts we should be able to minimise stall time, so time when the kernel is not doing anything and waiting for data to arrive. Ideally, every clock cycle, the kernel should be reading and writing data. By assuming there is no stall time at all, we can calculate the theoretical maximum bandwidth. We know that the maximum amount of data a port can write/read per clock cycle is equal to the width of the port times the clock speed. As the current clock speed is equal to the default 300 MHz the maximum amount of data written per second is equal to:

$$32\,\text{bit} \cdot 300\,\text{MHz} = 9.6\,\text{Gbit}\,\text{s}^{-1} = 1.2\,\text{GB}\,\text{s}^{-1} \tag{4.1}$$

The following schematic shows an interpretation of how the kernel is synthesized based on this AXI configuration. All three ports are connected to DDR memory and have a width of one item, so 32 bits. This means that at maximum, one computation can be done per clock cycle.

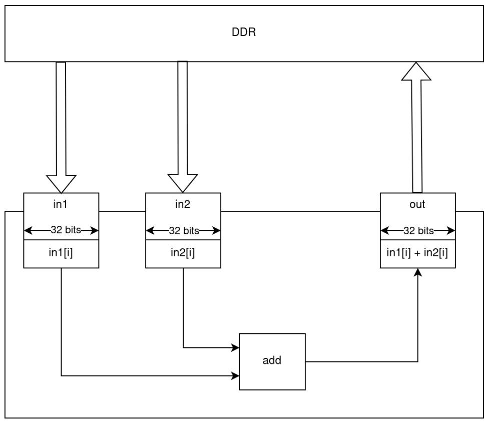


Figure 4.1: A schematic of the FPGA with 32-bit ports

While this is already a major improvement from the current performance, we can also apply port widening to increase this performance further.

Port widening

The maximum AXI port width can go up to 512 bits, which is 16 times as wide as our current width. This means that if we show the compiler that the FPGA consumes 16 items per clock cycle, we can increase the port width. Assuming a maximum clockspeed of 300 MHz, This would increase the theoretical maximum transfer speed for a single port to:

$$512\,\text{bit} \cdot 300\,\text{MHz} = 153.6\,\text{Gbit}\,\text{s}^{-1} = 19.2\,\text{GB}\,\text{s}^{-1} \tag{4.2}$$

The following schematic shows what this design would look like. Instead of reading or writing only one item per clock cycle, each port is able to read or write up to 16 items per clock cycle. This also means that we are able to perform 16 parallel calculations per clock cycle.

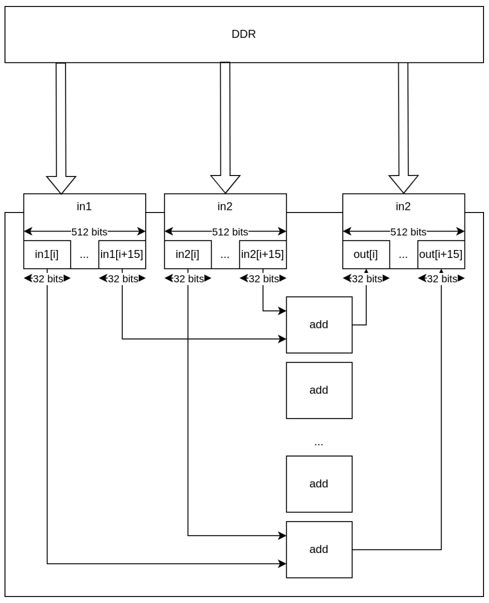


Figure 4.2: A schematic of the FPGA with 512-bit ports

4.2.2 Implementation

In attempt to achieve this design, we try three different implementations. All methods have been found in official examples or documentation.

By adding an assert statement By consulting the documentation we find that the HLS requirements for port widening are as follows [1]:

1. Must be a monotonically increasing order of access. You cannot access a memory location that is in between two previously accessed memory locations- aka no overlap.
2. The access pattern from the global memory should be in sequential order, and with the following additional requirements:
   (a) The sequential accesses need to be on a primitive type or non vector power of two size aggregate type
   (b) The start of the sequential accesses needs to be aligned to the widen word size
   (c) The length of the sequential accesses needs to be divisible by the widen factor

Most of these conditions are already fulfilled by our original definition of vector addition, as we already monotonically increase the order of access.

The only requirement missing is 2c: The compiler does not know if `size` is a multiple of our widen factor (32).
In order to fulfill this requirement, we can simply add an assert statement showing that the size should be divisible by 16:

```
# define PORT_WIDTH 16
...
void fpga_kernel(...) {
        assert(data % PORT_WIDTH == 0);
        for (int i = 0; i < size; i++) {
            out[i] = in1[i] + in2[i];
        }
    }
}
```

In order to enable burst for all ports we also add the following line to our compilation config file (`config.cfg`):

```
    [hls]
    ...
    syn.interface.m_axi_auto_max_ports=1
```

This enables creating multiple ports for each kernel argument. By default all kernel arguments are bundled into a single port, which means we use less FPGA resources, however this limits the total bandwidth to that of a single AXI port. So now we have 3 separate ports instead of 1.
From the compilation report we know that AXI burst and port widening has both been inferred from the source code.
When compiling and measuring the resulting kernel, we find a speed of $1.20\,\mathrm{GB\,s^{-1}}$ per port. This is a 30-fold improvement compared to the naive implementation, however is only 6.25% of the speed with respect to the theoretical maximum calculated in equation 4.2.
It can be seen that the speed is exactly equal to the theoretical maximum without port widening found in equation 4.1. From this we conclude that while port widening has been inferred, the actual calculation itself is not happening in parallel. The resulting circuit can be imagined as image 4.2, but instead of 16 parallel add blocks, there is only a single add block.
This means we need to rewrite our code such that the compiler infers the 16 parallel adds we want to accomplish.

By using a buffer array By looking at various examples in the Vitis example repository, we find a different approach [24]. Instead of inferring the parallel load and add, we can do this by adding a buffer and separating the read, calculate and write steps into separate for-loops:

```
...
  u_int buffer1[BURST_SIZE];
  u_int buffer2[BURST_SIZE];
  u_int bufferOut[BURST_SIZE];

```

```
outer:
  for (size_int i = 0; i < size; i += BURST_SIZE) {
      for (size_int b = 0; b < BURST_SIZE; b++) {
        buffer1[b] = in1[i + b];
      }
      for (size_int b = 0; b < BURST_SIZE; b++) {
        buffer2[b] = in2[i + b];
      }
      for (size_int b = 0; b < BURST_SIZE; b++) {
        bufferOut[b] = buffer1[b] + buffer2[b];
      }
      for (size_int b = 0; b < BURST_SIZE; b++) {
        out[i+b] = bufferOut[b];
      }
  }
...
```

The compiler will automatically flatten the inner for-loops as the iteration condition is a constant value (`BURST_SIZE`), this means that all iteration will be executed in a single clock cycle. In addition, the outer loop will be pipelined. Meaning that instead of executing the functions one after the other, components take a new input every iteration and pipeline the result to the next component:

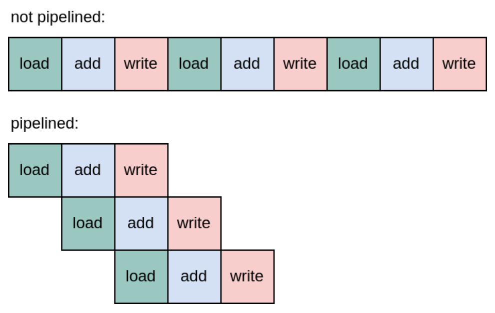


Figure 4.3: A schedule showing how tasks are executed, comparing the non pipelined execution order to the pipelined execution order

When synthesizing this code to a kernel binary and measuring the kernel we find that the port speed is 5.62 GB s^{-1}. This is a 140-fold improvement compared to the naive implementation, and 29.27% of the theoretical maximum calculated in equation 4.2.

By manually changing the port data types Another approach we find in the documentation and examples is using `hls::streams` and manually setting the data width to be 512 bits. This pattern is recommended by the Vitis documentation for burst inference. The advantage of this pattern is that process for transferring data to and from external memory is separated from the processing. In addition, port widening is forced as the data type is already 512 bits.

For our wide data type, we use the `hls::vector` type. In our case, we want to have a vector of 16 items, as this corresponds to the desired port width of 512 bytes.
In this implementation, we first read all the input data into input streams, then we add these streams and lastly we write the output stream to output. This design requires a different type of pipelining: Task level pipelining. This can be enabled using the dataflow pragma. Instead of pipelining operations, we now pipeline functions.
For example whenever we read a value using the `read_input` function and write it to `in_stream1`, this value can immediately be read by the `add_streams` function. This also means that the streams will always only contain one item, as the moment the `read_input` function puts an item into the stream, the `add_streams` function immediately reads this value in the next clock cycle.

```
# define WIDTH 16

typedef uint32_t size_int;
typedef float data_t;
typedef hls::vector<data_t, WIDTH> data_vec;
typedef hls::stream<data_vec> stream_t;

...

static void add_streams(...)  {
  for (size_int i = 0; i < size; i++) {
      data_vec in_value1 = in_stream1.read();
      data_vec in_value2 = in_stream2.read();

      out_stream << in_value1 + in_value2;
    }
}

extern "C" {
    void fpga_kernel(data_vec* in1, data_vec* in2, data_vec*
   out, size_int size) {
    assert(size % WIDTH == 0);
    const size_int stream_size = size / WIDTH;

# pragma HLS dataflow
    static stream_t in_stream1("in stream 1");
    static stream_t in_stream2("in stream 2");
    static stream_t out_stream("out stream");

    read_input(in1, in_stream1, stream_size);
    read_input(in2, in_stream2, stream_size);
    add_streams(in_stream1, in_stream2, out_stream, stream_size
   );
    write_result(out, out_stream, stream_size);
    }
}
```

When synthesizing and measuring this source code, this design results in exactly the same performance found in the buffer implementation. From this we conclude that the synthesized circuit from both designs is very similar.

4.2.3 Results

For these results we'll only look at kernel performance for 1 GB:

	Transfer speed (per port) ($\mathrm{GB\,s^{-1}}$)	Performance (GFLOPS)	Speedup
naive	4.00×10^{-2}	1.00×10^{-2}	-
assert	1.20	4.80×10^{-1}	30.00
buffer	5.62	1.41	140.50
manual	5.62	1.41	140.50

Overall we find that proper configuration of AXI has a big impact on the performance. We find that an just adding an assert statement, without modifying any other code, already results in a 30 fold speedup. However the compiler does not infer a possible parallel add, which creates a performance bottleneck of one computation per clock cycle.
If we do modify the code, by using a separate read, compute and write step, the compiler does create this parallel execution. This does require the programmer to write code which does not make much sense when looking at it from a software perspective, and requires them to understand the concept of pipelining. It does not make a difference in terms of performance which of the two methods we use, buffer or manual.
The performance is still only about a third of the theoretical maximum. When looking at the runtime reports, we find that there is still an external memory stall time of 70.67%. This means that about a third of the time, the kernel is waiting for transfers with external memory. This is an improvement from the naive version, but signifies that there is still improvement to be made in the port configuration.

4.3 Using multiple DDR cards

4.3.1 Design

When consulting the Vitis documentation on port configuration, we find that each DDR only has a single connection to the FPGA card, and a single connection can only read or write one item at a time.
This means that if we read 2 items from the same card this will happen sequentially instead of in parallel.
By default, all ports are connected to the same DDR card. So for each of the three ports (`in1`, `in2` and `out`), the reads and write happens sequentially. If instead we were to use two DDR cards, we would only have to read two data items sequentially, while the third would happen in parallel, resulting in a 1.5 times speedup.

When using three separate DDR cards, all three accesses happen in parallel, meaning we should get a 3 times speedup.

4.3.2 Implementation

This is implemented by adjusting the linking configuration file (`link.cfg`). In this configuration file we add the following lines of code

```
    nk=fpga_kernel:1:fpga_kernel
    sp=fpga_kernel.in1:DDR[0]
    sp=fpga_kernel.in2:DDR[1]
    sp=fpga_kernel.out:DDR[2]
```

Listing 4.2: The lines added to link.cfg file configured to use three DDR cards

Each input port gets explicitly assigned a different DDR card. The 4 DDR cards are identified by the names `DDR[0]` to `DDR[3]`.

We do not need to adjust the host or kernel code for this improvement.

4.3.3 Results

In our measurements, both the buffer implementation and the manual implementation perform the exact same. This is why we only show the results from the buffer implementation.

	kernel speed per port ($\mathrm{GB\,s^{-1}}$)	performance (GFLOPS)	speedup
1 DDR	5.62	1.41	140.50
2 DDR	8.89	2.22	222.25
3 DDR	6.91	1.73	172.75

It can be seen that for the 2 DDR the speedup is indeed by a factor of 1.5 compared to the 1 DDR version.

However when using 3 DDR cards, the speed actually decreases. This is because when compiling, the compiler was not able to achieve a frequency of 300 MHz. Instead the maximum frequency attained was 120.5 MHz.

This was the case for both the burst and manual implementation from section 4.2.

It's interesting to note that when using integers instead of floats as our data type, this did not happen and we got a performance of 15.9 $\mathrm{GB\,s^{-1}}$ per port. This is close to the maximum calculated bandwidth.

So the issue is likely caused by the combination of using DDR3, which is in a different SLR as the kernel, together with the increased resource usage from Floating Point Operations. As floating point operations are expensive to execute on FPGAs and can cause critical path issues. [10].

In order to make this improvement it was necessary to be aware of the general card structure, and the limitations of the DDR cards. The adjustments were only in configuration, and not dependent on code structure. However we did run into issues created by the critical path of the circuit design, something which we do not have much insight in on the HLS abstraction level, and can therefore be very unpredictable.

4.4 Multiple FPGA kernels

We find in the documentation that we can program multiple FPGA kernels onto the FPGA card.
An abstraction of this is shown in the following figure. Multiple instances of the same kernel are programmed onto the FPGA. We can run these instances in parallel.

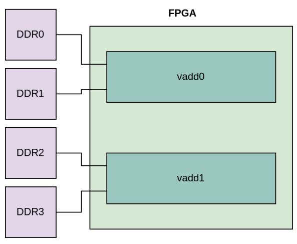


Figure 4.4: A schematic of the FPGA with 2 vector addition kernels

In order to implement this, we need to adjust the linking configuration file (`link.cfg`). We create two kernels in the shown configuration: `vadd0` and `vadd1`, then we can assign DDR cards separately to both of these kernels. This can be extended to using 4 kernels if we change the second line to `nk =fpga_kernel:4:vadd0,vadd1,vadd2,vadd3` and also add the DDR configuration for the third and fourth kernel.

```
[connectivity]
nk=fpga_kernel:2:vadd0,vadd1

sp=vadd0.in1:DDR[0]
sp=vadd0.in2:DDR[1]
sp=vadd0.out:DDR[0]

sp=vadd1.in1:DDR[2]
sp=vadd1.in2:DDR[3]
sp=vadd1.out:DDR[2]
```

Listing 4.3: The link.cfg file with two kernels

We also adjust the host code: We need to queue multiple kernels instead of one. In order to execute tasks in parallel, we also need to change the queue configuration to allow for concurrent execution, and add explicit dependencies between events so they do not happen out of order. We allocate data explicitly to the different DDR cards. Because if all data is allocated

to only one card, only the kernel with ports connected to that card is able to perform the execution.
Overall about a 100 lines of code need to be added or adjusted in order to accomplish this. See A.2 for the exact implementation details.
The kernel code does not need to change.
We try two implementations: Using two kernels, with a DDR mapping as shown in the code snippets above, and four kernels, where every kernel has all ports mapped to a single DDR port.

Results

We measure the results over different data sizes, as it could be that using more kernels causes a larger overhead.

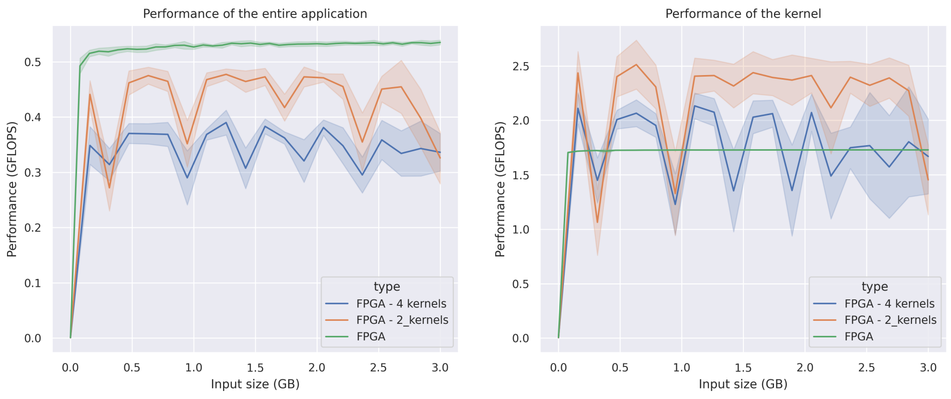


Figure 4.5: Performance of the vector addition application with multiple kernels

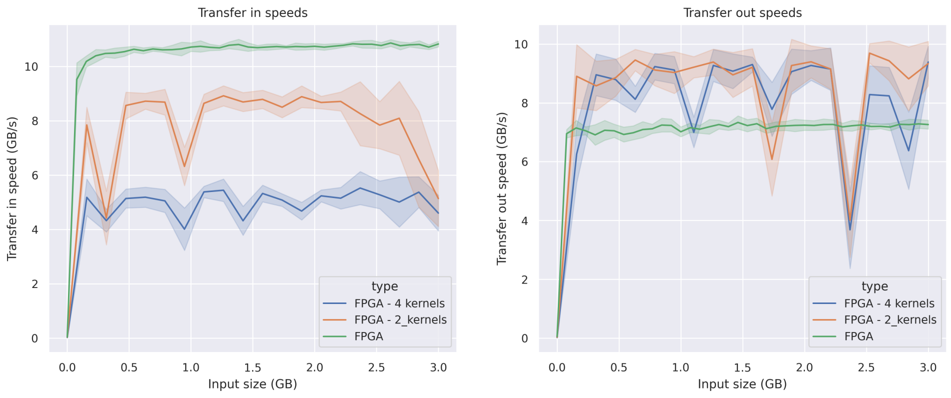


Figure 4.6: Transfer times of the vector addition application with multiple kernels

It can be seen that the overall implementation is slower compared to using a single kernel. This is because, while the kernel performance is higher for large data sizes, the transfer times are lower. Instead of transferring all the data in one go to a single DDR card, we need to divide the data over several DDR cards, this likely reduces the transfer speed.

4.5 Comparison to GPU and CPU

In the past sections we attempt various improvements, resulting in a more than 200 fold improvement in performance over the naive version. Now we want to see how this improved version compares against a CPU and GPU implementation.

4.5.1 Implementation

FPGA

For the FPGA we use the 2 DDR implementation described in 4.3.

CPU

For the CPU code we create a parallel for loop which is parallelized using an OpenMP pragma:

```
void vector_add(data_v& in1, data_v& in2, data_v& out) {
#pragma omp parallel for
  for (size_int i = 0; i < in1.size(); i++) {
    out[i] = in1[i] + in2[i];
  }
}
```

GPU

For the GPU we use the following kernel code. For each id, representing the index, the kernel checks if the id is within the bounds of the array. If it is, the kernel writes the sum of the input values at that index to the output vector.

```
const char *kernel_code =
__kernel void vector_add(
  __global float *a,
    __global float *b,
    __global unsigned int *c,
    const unsigned int n
) {
    int id = get_global_id(0);

    if (id < n)
        c[id] = a[id] + b[id];
}
```

Which we queue with a local size of 128.

4.5.2 Performance

Now that we have all three implementations, we measure the resulting applications in accordance to how they were described in the technical approach section 3.2.
From measurements we find the following results:

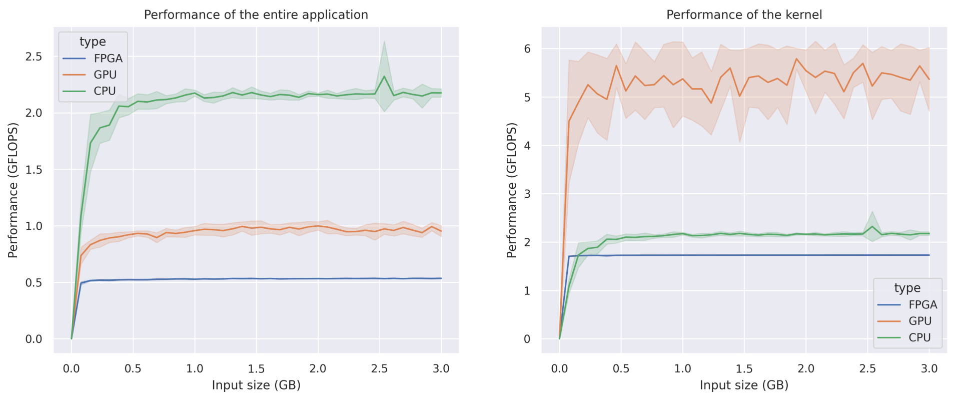


Figure 4.7: Performance of the vector addition applications of the GPU, FPGA and CPU

The CPU outperforms the GPU and FPGA if we take transfer times into account, this is because for both the FPGA and GPU, the transfer times are larger than the kernel times. For both these devices, the performance is bound by the speed between the host and the device global memory, so the PCI express bus. For the CPU this is not an issue, as the CPU uses the data directly from the host, so there are no transfer times.
As for kernel times, the GPU by far outperforms both the FPGA and the CPU. For all three devices, the performance is limited by memory. Because vector addition is arbitrary to compute, the computation is quicker than the memory bandwidth.
In this case, the FPGA does not have great performance compared to the other devices, this is different when we look at energy consumption.

4.5.3 Energy consumption

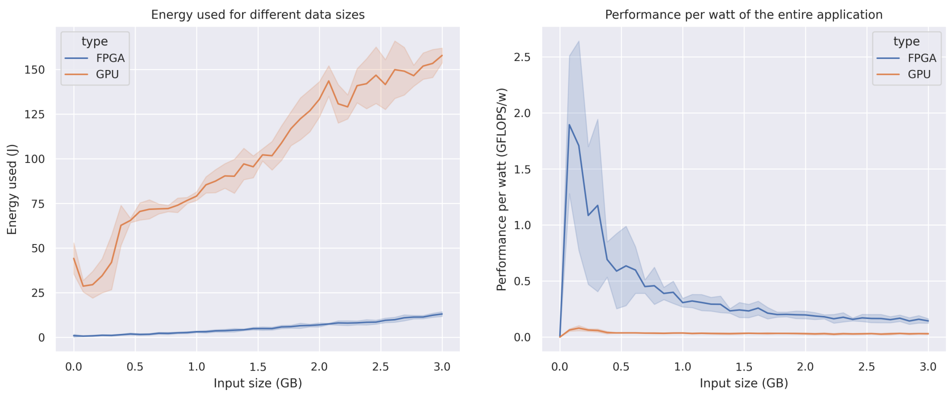


Figure 4.8: Energy consumption of the vector addition applications of the GPU and FPGA

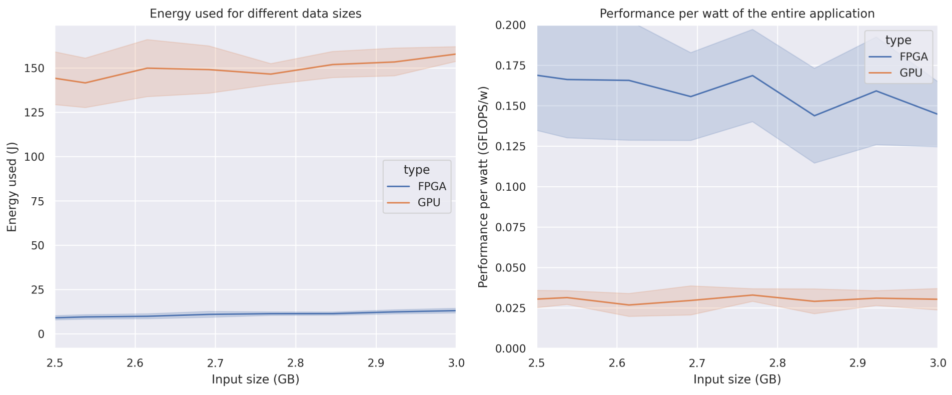


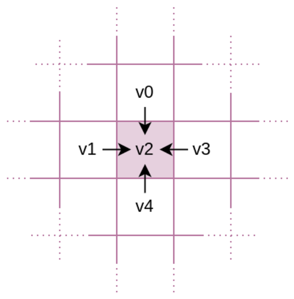
Figure 4.9: Energy consumption of the vector addition applications of the GPU and FPGA, zoomed in on the last 0.5 GB of measured data sizes

For our analysis we mostly look at the largest data sizes, as explained in 3.2.3. It can be seen that there is a spike in performance per watt for smaller data sizes for the FPGA, this is because the measurement method does not measure the wattage usage properly for short runs. The same holds, but on a smaller scale, for the GPU, shown by the small bump.
The energy consumption of the FPGA is significantly lower than the GPU. The FPGA uses about 15 times less energy than the GPU. When looking at performance per watt, the FPGA has a 6 times higher performance per watt.

Chapter 5

5-point stencil computation

A stencil computation is a computation which processes values in an array according to a predefined pattern. A 5-point stencil is a stencil which computes the new value of a point based on its direct neighbours. A visual representation of how the output value depends on its neighbours is shown in the image below.



Given two inputs: a 2-dimensional array, and a list of 5 coefficients, the program outputs a new 2-dimensional array. All output values which are not on the edges of the array are calculated as follows:

$$\begin{aligned} v_{x,y} = {} & c_0 \cdot v_{x-1,y-1} + c_1 \cdot v_{x,y-1} + c_2 \cdot v_{x+1,y-1} \\ & + c_3 \cdot v_{x-1,y} \end{aligned}$$

If a point is on the edge of the input, the output value is equal to the input value.

5.1 Naive implementation

We first implement this problem as simple as possible without any performance considerations.

The arguments for the kernel are as follows:

- `data_t* in`: the input data, this represents a 2-dimensional array containing the input data. The data has been flattened such that `in[y* width + x]` is equal to the value at position (x, y)
- `data_t* out`: the output data, formatted similar to `in`
- `data_t coefficients[5]`. This argument determines the coefficients for the stencil. Such that the top value is at index 0, the middle values 1-3 and the bottom value 4.
- `size_int width` is the width (and height) of the input array.

We have already configured the kernel to use separate ports and DDR cards.

The kernel code loops through all non-border cells and calculates the new values based on the neighbours of that cell.

```
typedef float data_t;
typedef uint32_t size_int;
# define STENCIL_SIZE 5

extern "C" {
    void fpga_kernel(data_t* in, data_t* out, data_t
    coefficients[STENCIL_SIZE], size_int width) {
    for (size_int y = 1; y < width - 1; y++) {
      for (size_int x = 1; x < width - 1; x++) {
        out[y*width + x] =
          coefficients[0] * in[(y-1)*width+x] +
          coefficients[1] * in[y*width+x-1] +
          coefficients[2] * in[y*width+x] +
          coefficients[3] * in[y*width+x+1] +
          coefficients[4] * in[(y+1)*width+x];
      }
    }
    }
}
```

The host code is similar to that of the vector addition code.

We synthesize this kernel code to a kernel binary and run the application. We find that while the program does synthesize to a kernel binary, when running it on the FPGA it does not return the correct results. The outer values at the edge are correct, but the inner values not.

Input:

```
0 1 2 3 4 5 6 7
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
60 61 62 63 64 65 66 67
70 71 72 73 74 75 76 77
```

Reference:

```
0 1 2 3 4 5 6 7
10 55 60 65 70 75 80 17
20 105 110 115 120 125 130 27
30 155 160 165 170 175 180 37
40 205 210 215 220 225 230 47
50 255 260 265 270 275 280 57
60 305 310 315 320 325 330 67
70 71 72 73 74 75 76 77
```

Output:

```
0 1 2 3 4 5 6 7
10 16711935 16711935 16711935 16711935 16711935 16711935 17
20 0 0 0 0 0 0 27
30 0 0 0 0 0 0 37
40 0 0 0 0 0 0 47
50 0 0 0 0 0 0 57
60 0 0 0 0 0 0 67
70 71 72 73 74 75 76 77
```

There are no warnings about correctness in the compile or link reports, and the code does return correct results when compiling and running it for software emulation.

5.2 Getting the code to work

In our debugging process, we explore various implementations to identify the root cause of the issue. Unfortunately, the compile reports provide limited guidance, making it challenging to pinpoint the exact problem.

A correct version that we found is based on the assumption that the cause of this issue is the DDR latency. In the AXI protocol, the kernel waits a fixed amount of cycles between requesting a value and assuming this value has arrived. We are attempting to request 5 values on the same port in a single iteration. Because the DDR can only handle a single request at a time, and the data is not sequential, the DDR latency is quite high. What can go wrong is that the latency of the DDR is higher than the waiting time, meaning that the FPGA will read incorrect data from the input ports.

We can solve this issue in two different ways: We can increase the latency, or we can decrease the amount of accesses to the DDR. We attempt the latter.

We try to decrease the amount of accesses for both ports: The `in` port and `coefficients` port.

5.2.1 Decreasing the amount of accesses to the coefficients port

We can decrease the amount of accesses to the coefficients port by saving the coefficients in a local buffer. We do this by creating a separate `load_coefficients` function which copies the values of an input array of size

STENCIL_SIZE to an output array. We then create a local array variable coefficients, and copy the values of the parameter in_c to this variable. In the computation code, we only read from coefficients, ensuring we don't access any values from the in_c port after the initial loading.

The resulting additions to the code are as follows:

```
void load_coefficients(const data_t in[STENCIL_SIZE], data_t
    out[STENCIL_SIZE]) {
load_coefficients_loop:
    for (size_int i = 0; i < STENCIL_SIZE; i++) {
        out[i] = in[i];
    }
}
...
static void main(.., data_t in_c[STENCIL_SIZE], ..) {
      ...
      data_t coefficients[STENCIL_SIZE];
      load_coefficients(in_c, coefficients);
      ...
}
...
```

5.2.2 Adding a cache to the in port

We find in the documentation that it is possible to add a cache to the input port. This means that the last items we have read from that input port are stored in a cache in the kernel. If we try to read this same item again it'll be directly returned by the cache instead of having to fetch the item from the DDR.

This means that for small data widths, possibly all previously accessed values can be read from the cache, and for large data widths, at least the three values next to each other can be read from the cache. We attempt a few random values for the "lines" and "depth" parameters of the cache and find that 16 and 16 are the higher values with which we can still synthesize our code. These parameters signify the amount of cache lines and the size of these lines respectively. These parameters can possibly be larger, however if the values are too large the linker first takes 9 hours to run before throwing an error. This means that it is not viable to try many different parameter combinations.

We enable the cache by adding an HLS pragma in the code:

```
...
static void main(data_t* in, ..) {
# pragma HLS cache port=in lines=16 depth=16
      ...
}
...
```

We do not need to adjust any other code for this.

After synthesizing kernel and measuring the application we find that the kernel does produce the correct results. When we measure the performance we find the following:

	size (GB)	time (s)	speed ($\mathrm{GB\,s^{-1}}$)	performance (GFLOPS)
alloc in	1.00	1.51×10^{-1}	1.33×10^{1}	-
kernel (per port)	1.00	1.55×10^{1}	6.46×10^{-2}	1.45×10^{-1}
alloc out	1.00	1.24×10^{-1}	8.04	-
total	1.00	1.58×10^{1}	6.35×10^{-2}	1.43×10^{-1}

The performance is initially already higher than the naive version of vector addition, this is because we already apply proper linking configuration here such as using multiple ports and DDR. However, similar to the performance of the naive version of vector addition, most of the time spent is in the kernel.

Based on what we know from the previous case, this is not surprising. We know from our last case that not using AXI burst has a large impact on performance, and burst is not enabled in this kernel as the data accessed is not sequential. This means that the biggest bottleneck is the transfer between the FPGA and the DDR.

5.3 Improvements by improving dataflow

5.3.1 Initial Design

The naive cache implementation has two big flaws: We do not access data sequentially, meaning we are not able to make use of AXI burst. In addition, we only use data fetched from DDR once, while this data is needed multiple times. First as upper neighbour, then three times as the three middle values and lastly as lower neighbour. As the application is now, we fetch this item again each time we need it, instead of keeping it in the kernel and reusing it.

In order to avoid these issues we can use a sliding window approach inspired by similar research which discusses implementing stencil computations on an FPGA: [8] and [23].

The "window" in this design consists of the 5 variables we need for the calculation: the upper value (v_0), the middle values (v_1-v_3) and the lower value (v_4). Every iteration we "slide" this window to the right to calculate the new value for the item at the position of v_2.

By using two FIFO queues we keep track of the intermediate values without having to read these again from global memory. `fifo_up` contains all the values in the input between v_0 and v_1, and `fifo_down` all those between v_3 and v_4. A visual representation of this structure can be seen in the image below.

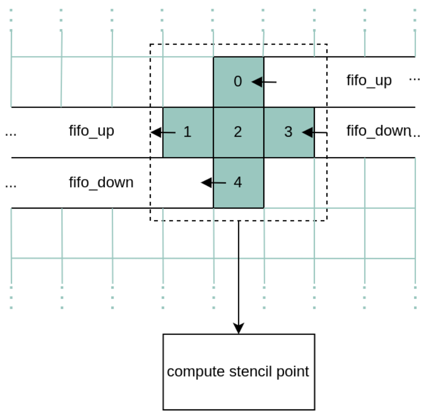


Figure 5.1: A visual representation of the sliding window structure

We can slide the window to the right as follows:

1. Write v_1 to `fifo_up`.
2. Write v_4 to `fifo_down`.
3. Shift the middle values one position to the left, so $v_1 = v_2$ and $v_2 = v_3$.
4. Read v_0 from `fifo_up`
5. Read v_3 from `fifo_down`
6. Read v_4 from the input.

We then send the values v0 to v4 to a compute unit, which is responsible for computing the result based on the current variables and the position.

The advantage is that all data is accessed sequentially, allowing for bursts, in addition, we only have to access each data point once, as it is reused.

This also comes with a disadvantage: We have to define a maximum width at compile-time. A FIFO must be able to contain all values between v0-v1, and v3-v4. This means that the FIFOs contain "width − 3" values at all times. These FIFOs must be allocated as memory components in the FPGA, which can't happen dynamically.

We limit the FIFO size to be $2^{14} = 16384$ items, which corresponds to a square array of a bit more than 1 GB. A larger size did not link due to space limitations of the FPGA.

5.3.2 Improvements by widening the dataflow width

The current design only allows us to process a single item per clock cycle. However we are able to read and write up to 16 items per clock cycle to global memory, as shown in the vector addition case. So our processing bandwidth should be 16 items per clock cycle.

We can accomplish this by widening the window size. Instead of sliding the window forward by one every time, we can slide the window by 16 items. This means that we can also calculate the output values for 16 items at a time instead of 1.

This means that v_0, v_2 and v_4 become lists instead of single values. We call these lists l_{up}, l_{mid} and l_{bot}.

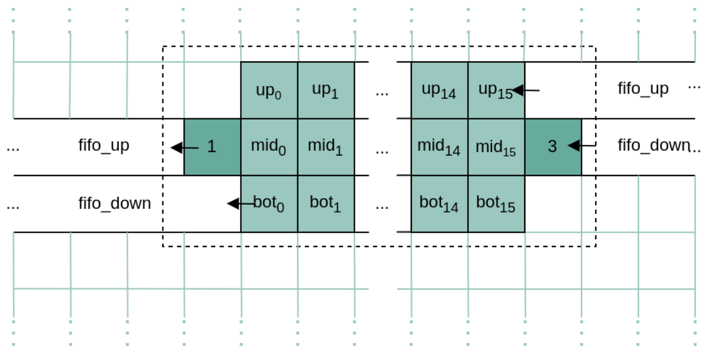


Figure 5.2: A visual representation of the wide sliding window structure

Every iteration we slide the window as follows:

1. Write v_1 and all values except the last one of l_{up} to `fifo_up`.
2. Write all of l_{bot} `fifo_down`.
3. Shift the middle values 16 items to the right, so $v_1 = l_{mid}[15]$ and $l_2[0] = v_3$.
4. Read l_{up} from `fifo_up`
5. Read all values except the first one l_{mid} and v_3 from `fifo_down`
6. Read l_{bot} from the input.

This can be shown visually as follows:

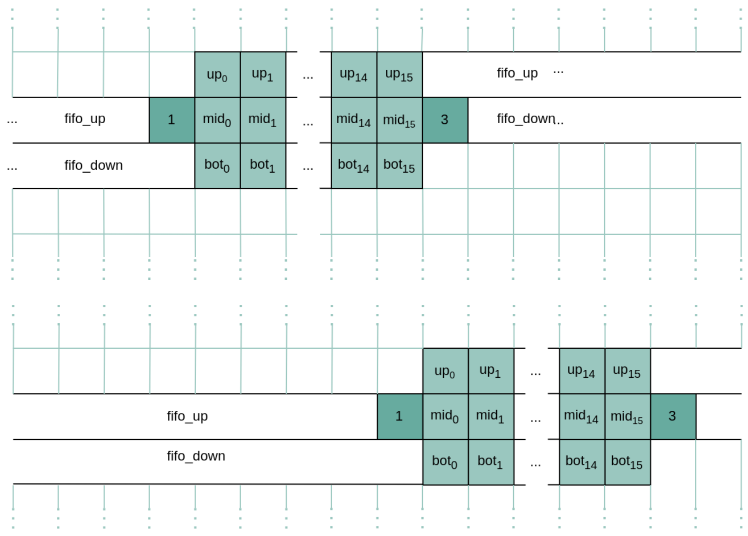


Figure 5.3: A visual representation of update of a wide sliding window structure

5.3.3 Implementation

Now that we have a design of the application, we need to implement this in Vitis HLS. For the full implementation, see code in the appendix A.3.

Instead of fixing the window width to be 16 values, we define a variable `P_UNROLL`, which defines the width of the window and with that, how many items we can process per iteration. In this text we call this value p_{unroll}.

Our programming pattern is similar to what we used in the buffer version of the vector add code. We have a loop where we read data into a buffer of size p_{unroll}, after that we compute the output and update the state, lastly, we write the result out to the output buffer.

Achieving Pipeline: II=1 The main loop is pipelined, this means that the length of the individual steps does not impact performance, as they can be executing in parallel. Ideally, you want a pipelined loop to process new inputs every single clock cycle, this is called a pipeline of `II=1`.

However, if the input of a step in a pipeline is also dependent on its output, meaning it is dependent on past loop iterations, the amount of clock cycles this step takes to execute matters. Every time we execute this step, we need to wait for the previous step to have finished, meaning that if the step takes multiple clock cycles, we are unable to achieve `II=1`.

For our code it is possible for the calculation function to be multiple clock cycles, as it is only dependent on the input, however the update function also has a state (the FIFOs). So in order to update, the previous update needs to be completed, this is why it is important that this function is able to execute in a single clock cycle.

The difference between the two functions is shown in the image below.

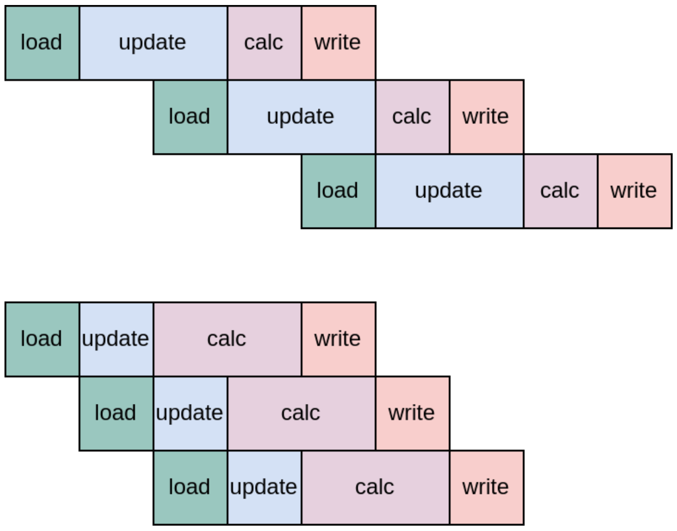


Figure 5.4: Two pipeline schemes, showing that if the update function takes multiple clock cycles this has impact on performance, but if the calc function does this is not the case

The update function The update function needs to shift the window one step to the right. It needs to be able to do this in a single clock cycle, It's implementation can be found in A.3 as the function `next_values`.

The update function writes the current window to the FIFOs, and then reads the next values into the window, both from the input and the FIFOs, as described in the algorithm design.

In order to implement the FIFOs we create a Fifo class: This class contains an array of size `MAX_WIDTH`, which is equal to the maximum width we can process. The class contains a "read" and "write" function which allows you to read or write p_{unroll} items from or to a buffer. These read and write functions need to be able to be executed in a single clock cycle.

In order to do this, we have attempted various methods. The final method which worked was to partition the array using the `PARTITION` pragma. Unlike the coefficients however, we don't fully partition the buffer but instead use a cyclic partition.

By default the array is implemented using only a single memory component (BRAM), however then it is only possible to read and write one item every clock cycle. In our case, we need to write and read p_{unroll} items.

To solve this we use the `PARTITION` pragma, this causes the array to be allocated to different components in a cyclic pattern.

For example if we partition the array `{0, 1, 2, 3, 4, 5}` with a factor of 3, values 0 and 3 belong to the first component, 1 and 4 to the second and 3 and 5 to the last.

We also attempted using the built-in `hls:stream` class. However we found that when trying to read and write from the stream in a single clock cycle, the compiler gave a warning that it was unable to achieve `II=1`.

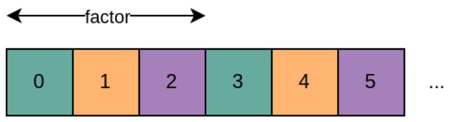


Figure 5.5: En example of a cyclic array partition with a factor of 3, every color represent an allocation to a different component

By partitioning the array with factor p_{unroll} we are able to write or read p_{unroll} sequential items in parallel. Which means that our `write` and `read` functions can be executed in a single clock cycle.

We ensure the write and read loops execute in a single clock cycle by using the unroll pragma. This pragma tells the compiler to execute each iteration of the loop in parallel.

Which means that our update function is also able to update in a single clock cycle.

The compute step The compute step is responsible for computing the output based on the current window and location.

As mentioned before, how many clock cycles this takes does not matter for performance. We find that if we do add an unroll pragma here to execute the add loop in a single clock cycle, we get a low frequency of around 120 Hz. This is why instead we use a pipeline pragma, to ensure the critical path does not get too long. The pipeline pragma forces the loop to be pipelined, which means that it is not possible to do all calculations in a single clock cycle.

```
data_t vs[5] = {v0, v1, v2, v3, v4};

data_t out_val = 0;
for (size_int i = 0; i < 5; i++) {
# pragma HLS pipeline
  out_val += coefficients[i] * vs[i];
}
```

Array partition We find that the compiler is unable to achieve pipeline `II=1` due to access problems because it is unable to read multiple values in one clock cycle from the arrays we use to store the window. In order to avoid this, we use the `PARTITION` pragma to fully partition the array. This means all values in the array are stored in separate registers and it is possible to read all these values in parallel in a single clock cycle.

Start handling At the start of the execution the FIFOs are completely empty. This means that we first need to read input into the FIFOs until they are filled with valid values. This takes $width + 2 \cdot p_{unroll}$ iterations. As the FPGA kernel needs to reads and write values in batches of p_{unroll}, and width doesn't necessarily need to be a multiple of p_{unroll}, we let this be handled by the host.

The host ignores the first $width+2 \cdot p_{unroll}$ output values from the FPGA.

End handling At the end of the buffer, we still want the window to slide forward but without reading any new inputs. To solve this, we let the host allocate extra data at the end of the input data.

Port mapping We map the in and out ports to DDR0 and DDR1 respectively. We map the coefficients port to PLRAM. This is a memory component of 128KB max embedded in the FPGA. With this we hope to prevent the frequency issues caused by using 3 DDR cards seen in 4.3 as the PLRAM is on the same SLR as where we place the kernel.

5.3.4 Results

When running the applications corresponding to our two designs, we find the following performance:

	speed per port ($\mathrm{GB\,s^{-1}}$)	performance (GFLOPS)	speedup
Cache	6.46×10^{-2}	1.45×10^{-1}	-
Window ($p_{unroll} = 1$)	7.62×10^{-2}	1.71×10^{-1}	1.18
Window ($p_{unroll} = 16$)	1.46×10^{1}	3.29×10^{1}	226.01

It can be seen that there is barely any speedup for the sliding window using $p_{unroll} = 1$. This is because the compiler was only able to achieve a frequency of 25.0 MHz for this implementation. This is very interesting, considering this was not an issue for the higher p_{unroll} version, which contains the exact same code except for the different value of the constant variable `P_UNROLL`. Our theory is that the compiler likely does not allocate memory somewhere in the circuit due to this, resulting in a longer critical path. However we find that there is no way to look further into this issue without looking into the compiled HDL code.

The sliding window with a p_{unroll} of 16 does manage to get a maximum frequency of 300 MHz, resulting in a 226-fold speedup compared to the cache version. The final performance is limited by the FPGA-to-DDR memory bandwidth, as with a higher bandwidth we would be able to increase p_{unroll} to increase the performance.

When looking at how much hand optimization and domain knowledge was required, we find that our initial naive version did not return correct results, and there was not much guidance from the compiler or any other analysis tool in the Vitis tool chain on how to fix this. Knowledge about the exact details of the AXI protocol turned out to be necessary to debug the issue. In order to get better performance out of the program, it was necessary to apply an FPGA specific algorithm, in order to implement this, it was necessary to use various pragmas and FPGA specific structures. When implementing it is necessary to constantly be aware of what the resulting circuit will look like.

Similar to the vector addition case, clock frequency has significant impact on the performance of one of our versions: $p_{unroll} = 1$. We find that it is difficult to predict, explain or prevent this issue when programming on the HLS abstraction level.

5.4 Comparison to GPU and CPU

Now that we have an improved FPGA application, we can compare its performance against a GPU and CPU implementation.

5.4.1 Implementation of CPU and GPU

CPU The CPU code is similar to the naive version of the FPGA kernel code, we loop through all inner values and calculate the output value based on the neighbours of the input value. We parallelise the outer for loop using OpenMP.

```
#pragma omp parallel for
  for (size_int y = 1; y < width; y++) {
    for (size_int x = 1; x < width; x++) {
      out[y * width + x] =
        in[(y - 1) * width + x] * coefficients[0]
        + ...
    }
  }
}
```

GPU For the GPU we use the following kernel code. If the current index is within the bound of the array and not on a border, we calculate the new value based on its neighbours. The GPU executes this kernel for each of the indices in the input.

```
__kernel void calc_stencil (...)
{
    unsigned int id_x = get_global_id(0);
    unsigned int id_y = get_global_id(1);

  if (id_x >= 1 && id_x < width - 1 && id_y >= 1 && id_y <
    width - 1) {
```

```
    out[id_y*width + id_x] =
            in[(id_y-1)*width+id_x]    * coefficients[0]
            + ...
  }
  else if (id_x >= 0 && id_x < width && id_y >= 0 && id_y <
    width) {
    out[id_y*width + id_x] = in[id_y*width + id_x] ;
  }
}
```

5.4.2 Performance

We run and measure the performance of the three applications.

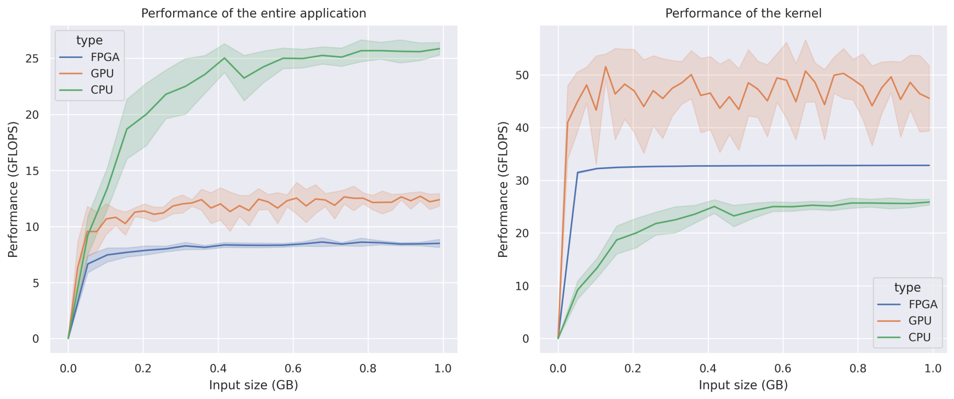


Figure 5.6: The performance metrics of the FPGA, GPU and CPU for the 5-point stencil computation

When looking at the entire application the FPGA performance stabilises at about 8 GFLOPS. The GPU at around 12-13 GFLOPS and the CPU at 25. For the FPGA and GPU, most time was spent on the transfer between host and global memory of the device.

The GPU performs best when looking at the kernel only. However, in comparison to the vector addition case, where the disparity was about 2.5 times, here the disparity is 1.6. This is because here we do not have issues with reduced clock speed, so we were able to use a higher percentage of the DDR bandwidth.

For all three implementations, the performance is higher compared to the performance of the vector addition. This is because more computations are done per memory read, this is especially the case for the FPGA as we have programmed it to reuse all read data. The limiting factor for the three applications remains to the memory accesses, as there are few computations compared to the memory speed.

Next, we look at the energy consumption of the FPGA compared to that of the GPU.

5.4.3 Energy consumption

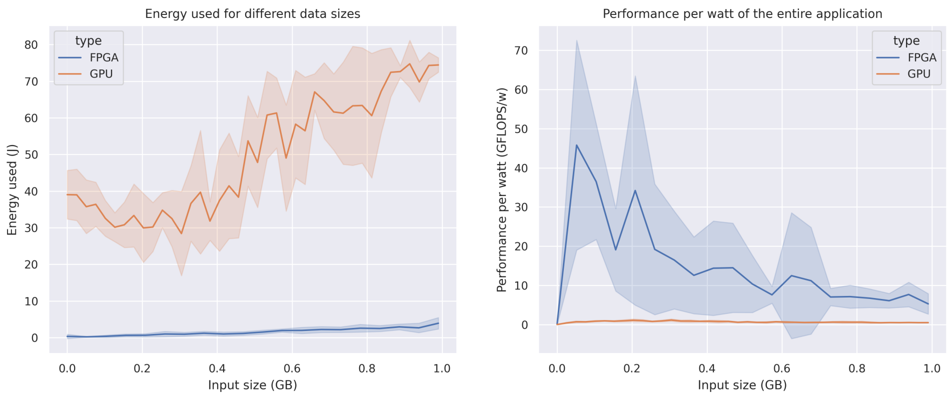


Figure 5.7: The energy consumption of the FPGA and GPU for the 5-point stencil computation

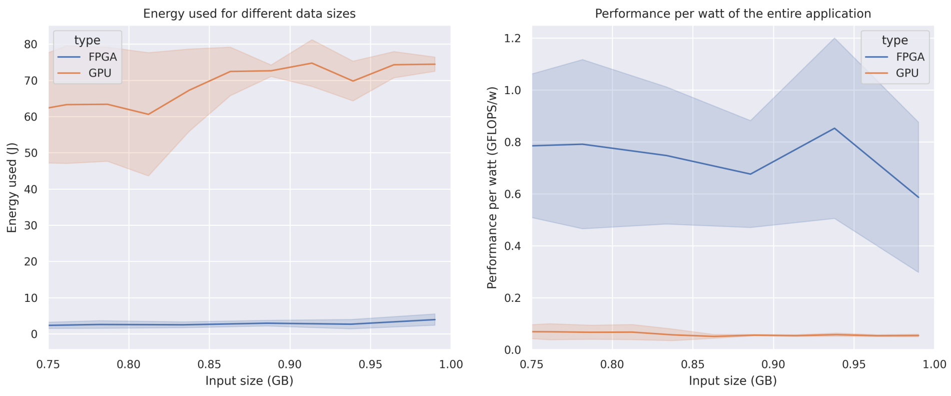


Figure 5.8: The energy consumption of the FPGA and GPU for the 5-point stencil computation, zoomed in on the last quarter of the measured data sizes

It can be seen that the FPGA consumes less energy compared to the GPU. Compared to the vector addition case the disparity in performance per watt is higher, this is because the relative performance is lower.

We are unable to get as much accuracy for the FPGA compared to the vector addition as we're unable to work with larger data sizes, meaning we have shorter execution times.

Chapter 6

Related Work

There exist several surveys on the overall development of high synthesis languages [21], [6], [17], [15]. These surveys look at possible tools and method of FPGA programming and compare and evaluate them. In contrast to this thesis, they are an overview of the overall state of FPGA development and highlight different tools. The Vitis tool chain we use is mentioned in [21] and [6]. [21] finds that while Vitis is a significant improvement over other methods and older tools, especially in regards to the system level integration, there is still domain knowledge required. This supports our findings.

Vitis is just one of various tools used to develop FPGAs. One category of tools being researched are on the same abstraction level as classic HDL tools such as VHDL but with more modern features such as parametrisation and polymorphism. [2] [3] [12]. As an example, one such language: Clash [2], compiles from Haskell to Verilog and VHDL. While the source language of these tools might be a similar or higher level of abstraction compared to the C++ used in our bachelor thesis, these languages are on a lower abstraction level in regards to circuit design. Other HLS tools on a higher abstraction level are also being developed [14] [16]. These languages are usually domain specific. For instance Lift [14] is a language focusing on functional patterns. The language most similar to Vitis is Intel's OpenCL HLS tool. We are not aware of research into the use of these other HLS languages for high performance computing, possible future work can be to repeat the approach of bachelor thesis but using one of these languages.

We are aware of one paper researching the development process using Vitis HLS specifically looking into the use process and domain knowledge: [4] shows the development process of a single application of Vitis HLS and has similar conclusions to this thesis. While this work is very similar, it does not take the same approach by starting with a naive version and comparing this to the hand optimised version.

Similar cases have been developed on FPGAs. [13] is an technical report of a floating point vector addition. [8] [25] and [19] all discuss implementing high performance stencil applications. Specifically [8] has inspired the design of the advanced stencil implementation in this thesis. The papers all get a higher end performance compared to our findings, this is due to a combination of a different problem scale, such as a 3d or chained stencil, and different hardware. Specifically the FPGA used in [8] has HBM memory which has higher memory bandwidth compared to the DDR memory used in our thesis. These papers have a different focus compared to this thesis, the goal is getting the best performance without looking at the naive implementation or the development process itself.

Chapter 7

Conclusions

Both case studies have demonstrated that a naive implementation at best leads to a significantly worse performance, with more than a 200-fold difference between the naive implementation and the hand optimised version. This is a surprising observation, specifically for vector addition as it is such a simple, common problem. The naive implementation of the stencil computation shows us that naive implementations can also not work at all. We conclude that it is not viable yet to write naive code when programming FPGAs using HLS, especially when performance is an important factor.

For both cases, we find that hand optimisations do require domain knowledge in order to work, especially knowledge about the general structure of the card, the memory access protocol and general good programming practices for Vitis, such as the read-compute-store pattern.

For the stencil computation specifically, we find that it is necessary to first design a high-level circuit design, which can then be translated into the HLS language. It necessary to always be aware of clock cycles, memory division and pipelining.

In conclusion, while HLS tools make it significantly easier to program FPGAs compared to HDL tools, there is still domain knowledge required on how hardware design works and the developer must be aware of the circuit which is synthesized.

When looking at the final performance of the optimised FPGA kernels we find that for both cases we hit a memory bottleneck: the limiting factor was the DDR memory bandwidth. In addition, when looking at the entire application, most of the time spent was in the data transfer between the host and global memory. In neither of the cases did we hit a limit in FPGA computing resources themselves (LUT, DSPs), we suspect that for more complicated problems the FPGA kernel can be fully utilised and achieve higher performance levels.

If we look at the comparison to non optimised GPU and CPU code we find that for both cases we hit memory bottlenecks on all devices. The CPU performed best when looking at the entire application, and the GPU best when looking at only the kernel. The FPGA had the lowest performance compared to other devices in the vector addition application. But when looking at the kernel speed of the stencil did perform better than the CPU.

When looking at energy usage compared to a GPU, the FPGA used significantly less energy and has a higher performance per watt. This is only possible from carefully optimised HLS code, and the development process was significantly more complicated than that of the GPU and CPU.

7.1 Practical observation

7.1.1 Technical installation and use process

We find that Vitis is difficult to install and use. First of all, the whole program is more than 200 GB large. This includes a user interface, a compiler for both normal HLS and AI specific HLS, a linker, a simulator, and a debugger. It is not possible to just install the parts which are necessary, you have to install all of it.

Vitis does provide a GUI which includes a way to compile projects, however there's not a documented way to compile projects made by the GUI using the terminal. So this was not viable when using slurm. Instead we wrote CMake files and used a self-configured text-editor, which worked well. However this means we might have missed out on some Vitis features by using this method.

For users, this would be something to take into consideration, as installing the GPU and CPU development tools and compilers is much easier and takes less space.

One other issue is that the documentation of Vitis can be pretty unclear. There are a lot of dead links in the documentation, links which lead to an error or empty page. In addition, a lot of the examples don't use optimal code practices for performance.

This can also be found in the results of research papers: For example [8] used a manual conversion from a 512 bit integer to widen to ports, while they could have used a `hls::vector`. This is likely because the documentation for the vector is hidden somewhere deep in the documentation and not used in any examples provided.

7.1.2 Compilation times

As mentioned in my thesis, the compilation times when linking the kernel can take a lot of time and memory. The times could go from 1 hour up to 10+ depending on the kernel. In addition, at least 32 GB of memory is required. When deciding whether to use an FPGA, this should also be taken into account.

The compilation times for GPU and CPU did not take more than 10 seconds Assuming the best case and for the FPGA and the worst case for the GPU/CPU the FPGA compilation time is 360 times as slow. When developing many different programs, this should be taken into account. Although if the program is ment to be reused for many times, this will have less of an impact on the choice.

One additional disadvantage is that this makes optimising very difficult. For example, the maximum frequency is only known after the linking has completed, so debugging low frequencies is difficult as for each attempt, you need at least an hour to find out the result. In some cases, such as the cache with high values, the compiler would not be able to complete because it took more than 10 hours to compile the program, which is the maximum set in the slurm configuration used.

It's important to note that these two applications were also relatively small, and the documentation mentions that compilation times are based on the source. So it is possible that this might be even more of an issue for larger projects.

7.2 Future work

Cases In both my cases, we did not take full advantage of the pipelining possible by an FPGA, this would likely lead to better performance of the FPGA compared to the GPU and CPU.

Possible cases which could be explored would be a repeated stencil computation, or signal or image processing problems.

In addition, in both cases we hit the memory bandwidth limit. It would also be interesting to explore cases where this harder or impossible to accomplish, such as a matrix multiplication. For these cases, techniques such as using multiple kernels could lead to a bigger improvement.

Optimizing clock frequency We have not looked much into how to optimise the clock frequency and prevent long critical paths. Nor did we look into what exactly causes extremely low frequencies and how to prevent this when using HLS languages.

Comparison to other HLS tools It would be interesting to explore these cases by using different languages to program the FPGA kernel. Using Vitis, it is possible to use custom RTL kernels which can then be linked.

Future research could explore implementing these problems using different languages and tools and seeing how the code, energy usage and performance compare.

Bibliography

[1] *Alveo Product Details • Alveo U200 and U250 Data Center Accelerator Cards Data Sheet (DS962) • Reader • AMD Technical Information Portal.* URL: `https://docs.amd.com/r/en-US/ds962-u200-u250/Alveo-Product-Details` (visited on 06/05/2024).

[2] Christiaan Baaij et al. "C?aSH: Structural Descriptions of Synchronous Hardware Using Haskell". In: *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools.* DSD '10. USA: IEEE Computer Society, Sept. 1, 2010, pp. 714–721. ISBN: 978-0-7695-4171-6. DOI: `10.1109/DSD.2010.21`. URL: `https://doi.org/10.1109/DSD.2010.21` (visited on 10/02/2023).

[3] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference.* DAC '12. New York, NY, USA: Association for Computing Machinery, June 3, 2012, pp. 1216–1225. ISBN: 978-1-4503-1199-1. DOI: `10.1145/2228360.2228584`. URL: `https://doi.org/10.1145/2228360.2228584` (visited on 07/29/2024).

[4] Nick Brown. "Weighing Up the New Kid on the Block: Impressions of Using Vitis for HPC Software Development". In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL).* 2020 30th International Conference on Field-Programmable Logic and Applications (FPL). Aug. 2020, pp. 335–340. DOI: `10.1109/FPL50879.2020.00062`. URL: `https://ieeexplore.ieee.org/abstract/document/9221613?casa_token=rdeoF2HIGpsAAAAA:MI97y4NQyz9HfXYJGBSIFqp6qlCXhOqx7M4E63LzFVwNuWexVk3hPX6E6XDZgJivtwj6LvaJabRDfFw` (visited on 05/28/2024).

[5] *Buffer Creation and Data Transfer • Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) • Reader • AMD Technical Information Portal.* URL: `https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration/Buffer-Creation-and-Data-Transfer` (visited on 06/06/2024).

[6] Jason Cong et al. "FPGA HLS Today: Successes, Challenges, and Opportunities". In: *ACM Transactions on Reconfigurable Technology*

and Systems 15.4 (Aug. 8, 2022), 51:1–51:42. ISSN: 1936-7406. DOI: 10.1145/3530775. URL: https://dl.acm.org/doi/10.1145/3530775 (visited on 05/28/2024).

[7] Tomasz S. Czajkowski et al. "From Opencl to High-Performance Hardware on FPGAS". In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 22nd International Conference on Field Programmable Logic and Applications (FPL). Aug. 2012, pp. 531–534. DOI: 10.1109/FPL.2012.6339272. URL: https://ieeexplore.ieee.org/document/6339272 (visited on 10/05/2023).

[8] Changdao Du and Yoshiki Yamaguchi. "High-Level Synthesis Design for Stencil Computations on FPGA with High Bandwidth Memory". In: *Electronics* 9.8 (8 Aug. 2020), p. 1275. ISSN: 2079-9292. DOI: 10.3390/electronics9081275. URL: https://www.mdpi.com/2079-9292/9/8/1275 (visited on 04/01/2024).

[9] *HLS Pragmas • Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) • Reader • AMD Technical Information Portal.* URL: https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration/HLS-Pragmas (visited on 08/11/2024).

[10] *Integer vs. Floating-Point Processing on Modern FPGA Technology — IEEE Conference Publication — IEEE Xplore.* URL: https://ieeexplore.ieee.org/abstract/document/9031118?casa_token=88cYsmJu4OoAAAAA:Gs-HNkfNDwlBmC6FmFMtxLH-wLk3l0qE-Wg7LTmCwYGTop4JVsjPtT1GTeaxQ
eJWN0vU (visited on 07/01/2024).

[11] Amin Isazadeh, Davide Ziviani, and David E. Claridge. "Global Trends, Performance Metrics, and Energy Reduction Measures in Datacom Facilities". In: *Renewable and Sustainable Energy Reviews* 174 (Mar. 1, 2023), p. 113149. ISSN: 1364-0321. DOI: 10.1016/j.rser.2023.113149. URL: https://www.sciencedirect.com/science/article/pii/S1364032123000059 (visited on 07/27/2024).

[12] Keerthan Jaic and Melissa C. Smith. "Enhancing Hardware Design Flows with MyHDL". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. New York, NY, USA: Association for Computing Machinery, Feb. 22, 2015, pp. 28–31. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689092. URL: https://doi.org/10.1145/2684746.2689092 (visited on 07/29/2024).

[13] Zheming Jin et al. *Evaluation of the Single-precision Floatingpoint Vector Add Kernel Using the Intel FPGA SDK for OpenCL*. ANL/ALCF-17/2. Argonne National Lab. (ANL), Argonne, IL (United States), Apr. 20, 2017. DOI: 10.2172/1357902. URL: https://www.osti.gov/biblio/1357902 (visited on 06/30/2024).

[14] Martin Kristien et al. “High-Level Synthesis of Functional Patterns with Lift”. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2019. New York, NY, USA: Association for Computing Machinery, June 8, 2019, pp. 35–45. ISBN: 978-1-4503-6717-2. DOI: `10.1145/3315454.3329957`. URL: `https://dl.acm.org/doi/10.1145/3315454.3329957` (visited on 10/02/2023).

[15] Sakari Lahti et al. “Are We There Yet? A Study on the State of High-Level Synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (May 2019), pp. 898–911. ISSN: 1937-4151. DOI: `10.1109/TCAD.2018.2834439`. URL: `https://ieeexplore.ieee.org/abstract/document/8356004?casa_token=vIGwHjk37IUAAAAA:7kRMJZcdkAlsF_krYFrnsFXaJXouGHFW3xXutF8-En9KdYsEOdabNfVL0V08luKmSNGCMYky7fMecwo` (visited on 05/27/2024).

[16] Yi-Hsiang Lai et al. “HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’19. New York, NY, USA: Association for Computing Machinery, Feb. 20, 2019, pp. 242–251. ISBN: 978-1-4503-6137-8. DOI: `10.1145/3289602.3293910`. URL: `https://doi.org/10.1145/3289602.3293910` (visited on 07/29/2024).

[17] Zhengjie Li et al. “A Survey of FPGA Design for AI Era”. In: *Journal of Semiconductors* 41.2 (Feb. 2020), p. 021402. ISSN: 1674-4926. DOI: `10.1088/1674-4926/41/2/021402`. URL: `https://dx.doi.org/10.1088/1674-4926/41/2/021402` (visited on 06/05/2024).

[18] *OpenCL Programming • Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) • Reader • AMD Technical Information Portal*. URL: `https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration/OpenCL-Programming` (visited on 07/24/2024).

[19] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. “Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.3 (Mar. 2014), pp. 695–705. ISSN: 1558-2183. DOI: `10.1109/TPDS.2013.51`. URL: `https://ieeexplore.ieee.org/abstract/document/6470606?casa_token=6ksDen-u_J4AAAAA:8xbzOWPciTDiaOW1QieBqTbYmhgk5jrhc`
`EScGemtWdV7Ba7AdQ1HrRA3tURbh58BkiI3bcRpjPOLA` (visited on 07/29/2024).

[20] Scott Sirowy and Alessandro Forin. “Where’s the Beef? Why FPGAs Are So Fast”. In: ().

[21] Emanuele Del Sozzo et al. “Pushing the Level of Abstraction of Digital System Design: A Survey on How to Program FPGAs”. In: *ACM Computing Surveys* 55.5 (Dec. 3, 2022), 106:1–106:48. ISSN: 0360-0300. DOI: `10.1145/3532989`. URL: `https://dl.acm.org/doi/10.1145/3532989` (visited on 10/02/2023).

[22] *Vitis Unified Software Platform Documentation Landing Page (UG1416) • Viewer • AMD Technical Information Portal*. URL: `https://docs.amd.com/v/u/en-US/ug1416-vitis-documentation` (visited on 07/24/2024).

[23] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. “Multi-FPGA Accelerator Architecture for Stencil Computation Exploiting Spacial and Temporal Scalability”. In: *IEEE Access* 7 (2019), pp. 53188–53201. ISSN: 2169-3536. DOI: `10.1109/ACCESS.2019.2910824`. URL: `https://ieeexplore.ieee.org/abstract/document/8689014` (visited on 02/22/2024).

[24] *Xilinx/Vitis_Accel_Examples: Vitis_Accel_Examples*. URL: `https://github.com/Xilinx/Vitis_Accel_Examples/tree/main` (visited on 06/30/2024).

[25] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. “Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL”. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’18. New York, NY, USA: Association for Computing Machinery, Feb. 15, 2018, pp. 153–162. ISBN: 978-1-4503-5614-5. DOI: `10.1145/3174243.3174248`. URL: `https://doi.org/10.1145/3174243.3174248` (visited on 06/30/2024).

[26] Xingqi Zou et al. “Breaking the von Neumann Bottleneck: Architecture-Level Processing-in-Memory Technology”. In: *Science China Information Sciences* 64.6 (June 2021), p. 160404. ISSN: 1674-733X, 1869-1919. DOI: `10.1007/s11432-020-3227-1`. URL: `https://link.springer.com/10.1007/s11432-020-3227-1` (visited on 06/17/2024).

Appendix A

Appendix

A.1 Example of host code

```
...
typedef uint32_t size_int;
typedef uint32_t data_t;
...

int main(int argc, char** argv) {
  // process arguments
  ...

    // Allocate Memory in Host Memory
    size_int vector_size_bytes = sizeof(data_t) * data_size;
    std::vector<data_t, aligned_allocator<data_t>> in1(
   data_size);
    std::vector<data_t, aligned_allocator<data_t>> in2(
   data_size);
    std::vector<data_t, aligned_allocator<data_t>> out(
   data_size);

    std::vector<data_t, aligned_allocator<data_t> > reference(
   data_size);

    // Create the test and reference data
  ...

    auto devices = xcl::get_xil_devices();

    auto fileBuf = xcl::read_binary_file(binaryFile);
    cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()
   }};
    auto device = devices[0];

    cl_int err;
    OCL_CHECK(err, cl::Context context(device, nullptr, nullptr
   , nullptr, &err));
    OCL_CHECK(err, CommandQueue q(context, device,
    CL_QUEUE_PROFILING_ENABLE, &err));

    std::cout << "Trying to program device " << device.getInfo<
    CL_DEVICE_NAME>() << std::endl;
    cl::Program program(context, {device}, bins, nullptr, &err)
    ;
    if (err != CL_SUCCESS) {
    std::cout << "Failed to program device with xclbin file!\n"
    ;
        exit(EXIT_FAILURE);
    } else {
    std::cout << "Device: program successful!\n";
    OCL_CHECK(err, cl::Kernel krnl_add(program, "fpga_kernel",
    &err));
    }

    // Allocate Buffer in Global Memory
    OCL_CHECK(err, cl::Buffer buffer_in1(
        context,
        CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
        vector_size_bytes,
        in1.data(),
              &err
  ));
    OCL_CHECK(err, cl::Buffer buffer_in2(
        context,
        CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
        vector_size_bytes, in2.data(),
                &err
  ));
    OCL_CHECK(err, cl::Buffer buffer_out(
        context,
        CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
        vector_size_bytes,
        out.data(),
        &err
  ));

    argc = 0;
    OCL_CHECK(err, err = krnl_add.setArg(argc++, buffer_in1));
    OCL_CHECK(err, err = krnl_add.setArg(argc++, buffer_in2));
    OCL_CHECK(err, err = krnl_add.setArg(argc++, buffer_out));
    OCL_CHECK(err, err = krnl_add.setArg(argc++, data_size));

    cl::Event in_event;
    cl::Event kernel_event;
    cl::Event out_event;

    Clock clock;
    clock.start();

    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_in1
    , buffer_in2}, 0, NULL, &in_event));

    OCL_CHECK(err, err = q.enqueueTask(krnl_add, NULL, &
   kernel_event));

    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_out
   }, CL_MIGRATE_MEM_OBJECT_HOST, NULL, &out_event))
    OCL_CHECK(err, err = q.finish());

  // verify data
  ...
  // calculate and print measurements
  ...
}
```

A.2 Host code with multiple kernels

```
// includes
...
# define DIVISION 4

// helper functions and classes, same as normal host code
...



int main(int argc, char** argv) {
  // load kernel, create reference data, similar to normal host
    code
  ...

    OCL_CHECK(err, cl::Context context(device, nullptr, nullptr
   , nullptr, &err));
    OCL_CHECK(err, cl::CommandQueue q(context, device,
   CL_QUEUE_PROFILING_ENABLE |
   CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err));

  // program device
  std::vector<cl_mem_ext_ptr_t> in1_exts;
  std::vector<cl_mem_ext_ptr_t> in2_exts;
  std::vector<cl_mem_ext_ptr_t> out_exts;

  for (uint i = 0; i < DIVISION; i++) {
    cl_mem_ext_ptr_t ext;  // Declaring two extensions for both
    buffers
    ext.flags  = i | XCL_MEM_TOPOLOGY; // Specify Bank0 Memory
   for input memory
    ext.obj = in1.data() + i * slice_size;
    ext.param = 0 ;
    in1_exts.push_back(ext);
  }
  for (uint i = 0; i < DIVISION; i++) {
    cl_mem_ext_ptr_t ext;  // Declaring two extensions for both
    buffers
    ext.flags  = i | XCL_MEM_TOPOLOGY; // Specify Bank0 Memory
  for input memory
    ext.obj = in2.data() + i * slice_size;
    ext.param = 0 ;
    in2_exts.push_back(ext);
  }
  for (uint i = 0; i < DIVISION; i++) {
    cl_mem_ext_ptr_t ext;  // Declaring two extensions for both
     buffers
    ext.flags  = i | XCL_MEM_TOPOLOGY; // Specify Bank0 Memory
   for input memory
    ext.obj = out.data() + i * slice_size;
    ext.param = 0 ;
    out_exts.push_back(ext);
  }

  std::vector<cl::Event> in_events ;
  std::vector<cl::Event> kernel_events ;
  std::vector<cl::Event> out_events ;
  std::vector<cl::Buffer> buffers_in1;
  std::vector<cl::Buffer> buffers_in2;
  std::vector<cl::Buffer> buffers_out;
  std::vector<cl::Kernel> kernels;

  for (size_int i = 0; i < DIVISION; i++) {

    std::cout << "creating buffers" << std::endl;
    OCL_CHECK(err,
      buffers_in1.push_back(cl::Buffer (
        context,
        CL_MEM_READ_ONLY | CL_MEM_EXT_PTR_XILINX |
   CL_MEM_USE_HOST_PTR,
        slice_size_bytes,
        &in1_exts.at(i),
        &err
      )
    ));

    OCL_CHECK(err,
      buffers_in2.push_back(cl::Buffer (
        context,
        CL_MEM_READ_ONLY | CL_MEM_EXT_PTR_XILINX |
   CL_MEM_USE_HOST_PTR,
        slice_size_bytes,
        &in2_exts.at(i),
        &err
      )
    ));
    OCL_CHECK(err,
      buffers_out.push_back(cl::Buffer (
        context,
        CL_MEM_READ_ONLY | CL_MEM_EXT_PTR_XILINX |
   CL_MEM_USE_HOST_PTR,
        slice_size_bytes,
        &out_exts.at(i),
        &err
      )
    ));
  }

  for (size_int i = 0; i < DIVISION; i++) {

    size_int arg = 0;
    OCL_CHECK(err, cl::Kernel kernel (program, "fpga_kernel", &err));
    OCL_CHECK(err, err = kernel.setArg(arg++, buffers_in1.at(i)));
    OCL_CHECK(err, err = kernel.setArg(arg++, buffers_in2.at(i)));
    OCL_CHECK(err, err = kernel.setArg(arg++, buffers_out.at(i)));
    OCL_CHECK(err, err = kernel.setArg(arg++, 0));
    OCL_CHECK(err, err = kernel.setArg(arg++, slice_size));
    kernels.push_back(kernel);
  }

    Clock clock;
    clock.start();
  for (size_int i = 0; i < DIVISION; i++) {
    cl::Event in_event;
    OCL_CHECK(err, err = q.enqueueMigrateMemObjects(
          {buffers_in1.at(i),
          buffers_in2.at(i)},
          0, NULL, &in_event)
    );
    in_events.push_back(in_event);
  }
  for (size_int i = 0; i < DIVISION; i++) {
    cl::Event kernel_event;
    OCL_CHECK(err, err = q.enqueueTask(kernels.at(i), &in_events, &kernel_event));
    kernel_events.push_back(kernel_event);
  }
  for (size_int i = 0; i < DIVISION; i++) {
    cl::Event out_event;
    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffers_out.at(i)}, CL_MIGRATE_MEM_OBJECT_HOST, &kernel_events, &out_events[0]));
    out_events.push_back(out_event);
  }
  q.finish();
  clock.stop();

  // verify data and print timing results
  ...
}
```

A.3 Stencil code

```
#include <iostream>

#include <hls_stream.h>
#include <sys/types.h>
#include <ap_int.h>
#include <cassert>
/* Coefficients should be seen as following:
 *     |x-1| x |x+1
 *     |---|---|---
 * y-1|    | 0 |
 * y  | 1 | 2 | 3
 * y+1|    | 4 |
*/

typedef uint32_t data_t;
typedef uint32_t size_int;
typedef hls::stream<data_t> stream_t;

# define P_UNROLL 16
// width of p_index should be 2 log P_UNROLL
# define STENCIL_SIZE 5
# define MAX_WIDTH 16384

const size_int SLICE_SIZE = P_UNROLL + 2;

class Fifo {
# define STREAM_SIZE 16384
  typedef ap_uint<14> index_t;
  index_t read_index = 0;
    index_t write_index ;
    data_t stream[STREAM_SIZE];

    public:
  Fifo(size_t size) {
    write_index = size;
  }

  void read(data_t buffer[P_UNROLL]) {
streams_read_loop:
# pragma hls ARRAY_PARTITION variable=stream cyclic factor=
    P_UNROLL
# pragma hls ARRAY_PARTITION variable=buffer complete
      for (size_int i = 0; i < P_UNROLL; i++) {
# pragma hls UNROLL
      buffer[i] = stream[(index_t) read_index + i];
      }
    read_index += P_UNROLL;
  }

  void write(const data_t buffer[P_UNROLL]) {
streams_write_loop:
# pragma hls ARRAY_PARTITION variable=stream cyclic factor=
    P_UNROLL
# pragma hls ARRAY_PARTITION variable=buffer complete
```

```
      for (size_int i = 0; i < P_UNROLL; i++) {
# pragma hls UNROLL
      stream[(index_t) write_index + i] = buffer[i];
      }
    write_index += P_UNROLL;
  }

  size_t size() {
    if (write_index < read_index) {
      return STREAM_SIZE - read_index + write_index;
    }
    return write_index - read_index;
  }
#ifndef __SYNTHESIS__
  void print_contents() {
    for (size_int i = 0; i < size(); i += 1) {
      std::cout << stream[(index_t) read_index + i] << " ";
    }
    std::cout << std::endl;
  }
#endif
};


static void load_coefficients(const data_t in[STENCIL_SIZE],
    data_t out[STENCIL_SIZE]) {
load_coefficients_loop:
# pragma hls ARRAY_PARTITION variable=out complete
    for (size_int i = 0; i < STENCIL_SIZE; i++) {
# pragma hls UNROLL
    out[i] = in[i];
    }
}

static data_t calc_stencil_point (
  const data_t coefficients[STENCIL_SIZE],
  const size_int width, const size_int iteration,
  data_t v0, data_t v1, data_t v2, data_t v3, data_t v4
) {
  size_int loc = iteration - width - P_UNROLL - P_UNROLL;
    const size_int y = loc / width;
    const size_int x = loc % width;

    // std::cout << "calculating stencil for (" << x << ", " <<
     y << ") with v0:" << v0 << " v1:" << v1 << " v2:" << v2 <<
     " v3:" << v3 << " v4:" << v4 << std::endl;

    if (y <= 0) v0 = 0;
    if (x <= 0) v1 = 0;
    if (y >= width - 1) v4 = 0;
    if (x >= width - 1) v3 = 0;

  data_t vs[5] = {v0, v1, v2, v3, v4};

  data_t out_val = 0;
  for (size_int i = 0; i < 5; i++) {
# pragma HLS unroll factor=3
    out_val += coefficients[i] * vs[i];
  }
  return out_val;
}

static void compute_output (
  data_t buffer[P_UNROLL],
    const data_t* coefficients,
    const size_int width, const size_int loc,
    const data_t top[SLICE_SIZE], const data_t mid[SLICE_SIZE],
     const data_t bot[SLICE_SIZE]
) {
# pragma hls ARRAY_PARTITION variable=buffer complete
compute_output_loop:
    for (size_int i = 0; i < P_UNROLL; i++) {
# pragma HLS UNROLL
      buffer[i] = calc_stencil_point(coefficients, width, loc+i
    , top[i+1], mid[i], mid[i+1], mid[i+2], bot[i+1]);
    }
}

#ifndef __SYNTHESIS__
static void print_status (
    Fifo& fifos_up, Fifo& fifos_down,
    size_int x, size_int y,
    data_t top[SLICE_SIZE], data_t mid[SLICE_SIZE], data_t bot[
    SLICE_SIZE]
){
    std::cout << "=====[ " << "x: " << x << " y: " << y << "
    ]=====" << std::endl
  << "fifos up size " << fifos_up.size() << std::endl
  << "fifos down size " << fifos_down.size() << std::endl;
  std::cout << "fifos up contents: " ;
  fifos_up.print_contents();
  std::cout << "fifos down contents: " ;
  fifos_down.print_contents();
    std::cout << "status registers: " << std::endl;
    std::cout << "top\t" ;
    for (size_int i = 0; i < SLICE_SIZE; i++) {
      std::cout << top[i] << "\t" ;
    }
    std::cout << std::endl;
    std::cout << "mid\t" ;
    for (size_int i = 0; i < SLICE_SIZE; i++) {
      std::cout << mid[i] << "\t" ;
    }
    std::cout << std::endl;
    std::cout << "bot\t" ;
    for (size_int i = 0; i < SLICE_SIZE; i++) {
      std::cout << bot[i] << "\t" ;
    }
    std::cout << std::endl;
}
#endif

static void next_values(data_t input[P_UNROLL], Fifo& fifos_up,
     Fifo& fifos_down, data_t top[SLICE_SIZE], data_t mid[
    SLICE_SIZE], data_t bot[SLICE_SIZE]) {
  // update the fifos
# pragma HLS ARRAY_PARTITION variable=top complete
# pragma HLS ARRAY_PARTITION variable=mid complete
# pragma HLS ARRAY_PARTITION variable=bot complete
# pragma HLS ARRAY_PARTITION variable=input complete
  fifos_up.write(mid);
  fifos_down.write(bot);

  // shift the values P_UNROLL spots to the right
  top[0] = top[SLICE_SIZE - 2];
  top[1] = top[SLICE_SIZE - 1];

  mid[0] = mid[SLICE_SIZE - 2];
  mid[1] = mid[SLICE_SIZE - 1];

  bot[0] = bot[SLICE_SIZE - 2];
  bot[1] = bot[SLICE_SIZE - 1];

  // let middle and top read from their respective fifos
  fifos_up.read(top + 2);
  fifos_down.read(mid + 2);
  for (size_int i = 0; i < P_UNROLL; i++) {
# pragma HLS unroll
    bot[i+2] = input[i];
  }
}

void read_buffer(const data_t* in, data_t buffer[P_UNROLL],
    const size_int i) {
# pragma HLS array_partition variable=buffer complete
  for (size_int j = 0; j < P_UNROLL; j++) {
# pragma HLS UNROLL
    buffer[j] = in[i + j];
  }
}

void write_buffer(data_t* out, const data_t buffer[P_UNROLL],
    const size_int i) {
# pragma HLS array_partition variable=buffer complete
  for (size_int j = 0; j < P_UNROLL; j++) {
# pragma HLS UNROLL
    out[i + j] = buffer[j];
  }
}
void calc_stencil(
    const data_t* in, data_t* out, const data_t in_c[
    STENCIL_SIZE],
    const size_int width
    ) {
  const size_t n_iterations = width * width + width + P_UNROLL
   + P_UNROLL;

  data_t in_buffer[P_UNROLL];
  data_t out_buffer[P_UNROLL];
  Fifo fifos_up(width - P_UNROLL - 2);
  Fifo fifos_down(width - P_UNROLL - 2);
  data_t top[SLICE_SIZE] = {};
  data_t mid[SLICE_SIZE] = {};
  data_t bot[SLICE_SIZE] = {};

  for (size_t i = 0; i < n_iterations; i += P_UNROLL) {
# pragma HLS pipeline II=1
    read_buffer(in, in_buffer, i);
# ifndef __SYNTHESIS__
  print_status(fifos_up, fifos_down, i % width, i / width, top,
     mid, bot);
# endif
    compute_output(out_buffer, in_c, width, i, top, mid, bot);
    next_values(in_buffer, fifos_up, fifos_down, top, mid, bot)
   ;
    write_buffer(out, out_buffer, i);
  }
}

extern "C" {
    void fpga_kernel(data_t* in, data_t* out, const data_t in_c
   [STENCIL_SIZE], const size_int width) {
  #pragma HLS INTERFACE m_axi port=in bundle=gmem0
  #pragma HLS INTERFACE m_axi port=out bundle=gmem1
  #pragma HLS INTERFACE m_axi port=in_c bundle=gmem2
    assert (width <= MAX_WIDTH);
    assert (width >= P_UNROLL * 2);

      data_t coefficients[STENCIL_SIZE];
# pragma HLS ARRAY_PARTITION variable=coefficients complete
      load_coefficients(in_c, coefficients);
    calc_stencil(in, out, coefficients, width);
  }
}
```