OPTIMIZATION OF SIMPLIFIED SHALLOW WATER OPENCL APPLICATION ON FPGA

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2019

By
Ruiqi Ye
10292968
Supervisor: Graham Riley
School of Computer Science

Content

Word Count: 21195 (Main body only)

List of Figures

List of Tables

List of Diagrams

Abstract

High-performance computing has attracted more and more attention due the increasing computation power needs in areas like machine learning, big data processing and analysis. Heterogenous systems that use GPGPUs as accelerators are common candidates for high-performance computing these days. However, the power consumption of GPGPU has become a significant problem when it comes to scalability, for example, to build a supercomputer that can perform exascale computing. Therefore, FPGAs, which have a better power efficiency and flexible hardware architecture, have become the new candidate of the accelerator of heterogenous systems.

A simplified shallow water application developed using OpenCL is implemented and optimized on a Xilinx FPGA in this project. A series of experiments that consist of overhead analysis and overhead minimization are conducted. An overhead analysis that divides overheads into five different categories is applied to the baseline version of the code first. Then a series of optimization methods including loop pipelining, loop unrolling, burst memory transfer and using on-chip BRAM as cache are applied to the baseline, based on the result of overhead analysis. A principle which describes how to apply the optimization methods to the baseline is proposed. Overhead analysis and overhead minimization are iterative processes, they only stop after certain requirements are met. Furthermore, an experiment that aims to prove the Xilinx emulator is unreliable in terms of execution time prediction and performance improvement indication is also conducted. Two methods of efficient FPGA programming and correct execution time acquisition during experiments are proposed as well.

A method of interpreting the latency and loop information provided in the Xilinx HLS tool report is explained. A simple scalability model is also proposed for experimental data analysis. Result shows a maximum speedup of around 45x is achieved on the main computation loop of the simplified shallow water application, compared to the baseline. However, the highly optimized, simplified shallow water application that runs on a mid-range FPGA is still significantly outperformed by a not highly optimized simplified shallow water application which runs on an entry-level GPGPU. Some observations on power consumption are also provided.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgement

I would like to express my gratitude to my supervisor Mr. Graham Riley for his invaluable support, motivation, guidance, and feedback throughout this project. His constant motivation and enthusiasm encouraged me to produce my best work.
I would also like to thank my family and friend for the constant support without which this work would not have been possible.

Glossary

Accelerator It is a hardware device that accelerate the computation. Common accelerators including GPGPU, FPGA and Intel Xeon Phi. Accelerator is part of the heterogenous computing system.

API Application Programming Interface. It is "a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of communication among various components." (Application programming interface, 2019)

BRAM Block RAM. It is a type of on-chip memory used on FPGA.

Core It is a hardware component that can execute instructions while load and store data between itself and memory.

CPU Central Processing Unit. It is a hardware that can perform arithmetic, logic, control and I/O operation.

CU Compute Unit. It refers to a collection of processing elements according to the OpenCL platform model. A work group is executed on one compute unit.

Data Parallelism It refers to a type of parallelism that allows different cores access and process different data simultaneously.

DSP Digital Signal Processor. It is a type of hardware that specialized in digital signal processing. In FPGA, DSP is responsible for the execution of floating-point operation. Parallel computing can also be conducted using DSP by utilizing the SIMD instructions.

FF Flip-Flop. It is a circuit that have two states and can store state information. Its state can be changed by applying signals on to it. Flip-Flops usually have at least one input and one output. Flip-Flop is one of the most important components of FPGA.

FPGA Field Programmable Gate Array. It is a programmable hardware which consists of processing system and programmable logic. Different kind of IP block will be generated within the programmable logic, based on the algorithm.

Global Work Item Size It refers to the number of work items of all work groups. In short, it is the total number of work items that executes one kernel.

GPGPU General Purpose Graphical Processing Unit. It is an accelerator which is originally designed for processing and displaying 2D and 3D computer graphics. It is called "general purpose" because it can be utilized to process general data efficiently thanks to its SIMD architecture.

HDL Hardware Description Language. It is a programming model that precisely describes the structure and behaviour of the circuits.

Heterogeneous Computing A computing method that use different types of hardware. For example, CPU plus accelerators.

Homogeneous Computing A counterpart of heterogenous computing, which refers to computation that use only one type of hardware. For example, CPU.

HLS High Level Synthesis. It is a process that transform the high-level programming language like Java and C++ to hardware description language.

HPC High-Performance Computing. It is a practice that focus on solving the computation-intensive or data-intensive tasks efficiently.

IC Integrated Circuit. It is a collection of circuits that integrated on a small piece of silicon.

IDE Integrated Development Environment. It is a type of software which aims to facilitate the process of software development for developers. In general, IDE includes source code editor, debugger, compiler and building tools.

II Initial Interval. It refers to the time needed between the execution of the first iteration and the second iteration.

Iteration Latency It refers to the time needed for one iteration to complete.

Kernel It refers to a function executed on one compute unit according to the OpenCL execution model.

Local Work Item Size It refers to the number of work items within one work group. It also refers to the number of dimensions as well as the magnitude of each dimension of one work group.

LUT Look Up Table. It is one of the most important components of FPGA. Look up table is a truth table that store the results of Boolean operations, based on the inputs. It makes the execution of Boolean operation more efficient because the result can be obtained simply by checking the loop up table.

Memory Latency It refers to the time needed for accessing external, off-chip memory.

Multi-core It is a type of hardware that consists of multiple cores, for example, multi-core CPU.

NDRange It is an index space that describes the total number of work items which execute the kernel. The NDRange can either be 1-Dimensional, 2-Dimensional or 3-Dimensional.

OpenCL Open Computing Language. It is a programming model which is widely used in heterogenous computing. It can produce code that can be executed on CPU, GPGPU and FPGA.

PCIe Peripheral Component Interconnect express. It is a standard for high-speed serial computer bus interface. Hardware like accelerators and hard drives can be connected using the PCIe on motherboard.

PE Processing Element. A counterpart of core. According to the OpenCL platform model, multiple processing elements are included in one compute unit

Start Interval It refers to the time between the invocation of the first function call and the second function call.

Trip Count It refers to the size of iteration space.

VHDL Very High-Speed Integrated Circuit Hardware Description Language. It is one of the major HDL.

Vivado It is an HLS tool provided by Xilinx, which is integrated within SDx.

Work Group It refers to a collection of work items that are executed on one compute unit, in the OpenCL programming model.

Work Item It is equivalent to thread according to the OpenCL programming model.

SDAccel Software-Defined Accelerator. It is an IDE that targets the application development for Xilinx accelerators.

SDSoC Software-Defined System on Chip. It is an Eclipse-based IDE that targets the embedded C/C++/OpenCL application development for heterogenous Zynq SoC and MPSoC (Multi-Processor SoC) system.

SDx It is an Eclipse-based IDE provided by Xilinx.

SIMD Single Instruction, Multiple Data. An architecture that allows different cores access different data but process them in the same way simultaneously. In short, different data will be processed using the same instruction on different core simultaneously. SIMD is common in GPGPU.

Task Parallelism It refers to a type of parallelism that allows different cores execute different tasks simultaneously.

Chapter 1 Introduction

As we have entered the era of big data, high-performance computing has become increasingly important, since nowadays computation-intensive and data-intensive applications like weather forecasting must be computed within a limited amount of time. One popular way of achieving high-performance is “going parallel”, meaning using processors with multiple cores instead of the one with single core, because it is more and more difficult to achieve higher performance on a single core by simply increasing the clock frequency. However, even if the performance is achieved, the power consumption will be intolerable.

It has been found that homogenous multicore architectures, for example multicore CPU, are not the best candidate for high-performance computing. Multicore CPU is suitable for executing control-dominant problems, but it does not perform well on problems that need only little control flow and synchronization with other threads or tasks, compared to GPGPU. Therefore, heterogeneous architecture, for example CPU-GPGPU, becomes popular these days, because it can utilize the advantages of different homogeneous architectures while hiding most of their disadvantages.

In a heterogeneous system, devices like GPGPU, FPGA and Intel Xeon Phi are called accelerators. In this project, FPGA is chosen as the accelerator instead of GPGPU. This is because the hardware architecture of FPGA is more flexible, and the energy efficiency of FPGA is better compared to CPU and GPGPU, this is also mentioned in the work of (Georgopoulos et al., 2019). In addition, recent developments have made FPGAs easier to program.

This project aims to implement and optimize a simplified shallow water application developed using OpenCL, on an ARM CPU-FPGA heterogenous system. A series of optimization methods selected from literatures are applied to the code one after another following a certain principle, which will be discussed later. Performance estimation as well as overhead analysis are be conducted. Other contributions of this project including a method of efficient FPGA programming, a way of measuring correct and accurate execution time of the kernels executed on the FPGA, a detailed explanation regarding

how to interpret the latency and loop information given in the system estimate report and HLS report produced when building a FPGA solution, and a model to evaluate the scalability of each optimization method.

This dissertation is organized as follows. Chapter 2 provides some backgrounds on FPGA, OpenCL (the programming language used), SDx (the Xilinx tool with the SDSoC environment supporting FPGA development), stencil computation (as used in the simplified shallow water application) and optimization methods for FPGA applications developed using OpenCL. The literature review is also included. The research methodology is presented in Chapter 3, where a description regarding how the project is implemented is provided. Chapter 4 presents the description of a series of experiments with basic optimization methods. Chapter 5 describes a series of experiments with more advanced optimization methods. Experimental data are analysed in Chapter 6. Finally, Chapter 7 concludes the whole dissertation and identifies some future works.

1.1. Project Aims

The aims of this project are to implement a simplified but high-performance shallow water weather & climate forecasting application using OpenCL on an ARM CPU-FPGA heterogeneous system, while trying to program the FPGA efficiently from a high-level programmer's perspective.

1.2. Project Objectives

In order to achieve these aims, the following objectives must be accomplished, one after another.

- Understand FPGA programming using OpenCL and SDSoC of the Xilinx FPGA available for this project.
- Getting familiar with the HLS tools which makes FPGA programming easier.
- Develop a methodology which includes performance estimation, overhead analysis, overhead minimization, execution time acquisition, a principle for applying optimization methods and a method for efficient FPGA programming.
- Implement a shallow water weather & climate forecasting application using

OpenCL and execute it on a single FPGA, then apply the methods to obtain high performance.

Chapter 2 Background

This chapter introduces the essential background knowledge that is necessary for this project. Section 2.1 provides a general description of FPGA. Section 2.2 introduces the OpenCL programming language, including its platform model, execution model, kernel programming model and memory model. In Section 2.3, the IDE used in this project, called SDx, is introduced. Section 2.4 describes the principle of stencil computation, along with the basic idea of the shallow water weather & climate forecasting model, and the simplified shallow water application used in this project. Section 2.5 summarizes a series of optimization method for FPGA application developed in OpenCL. Section 2.6 provides a literature review of the papers that are key to this project.

2.1. FPGA Background

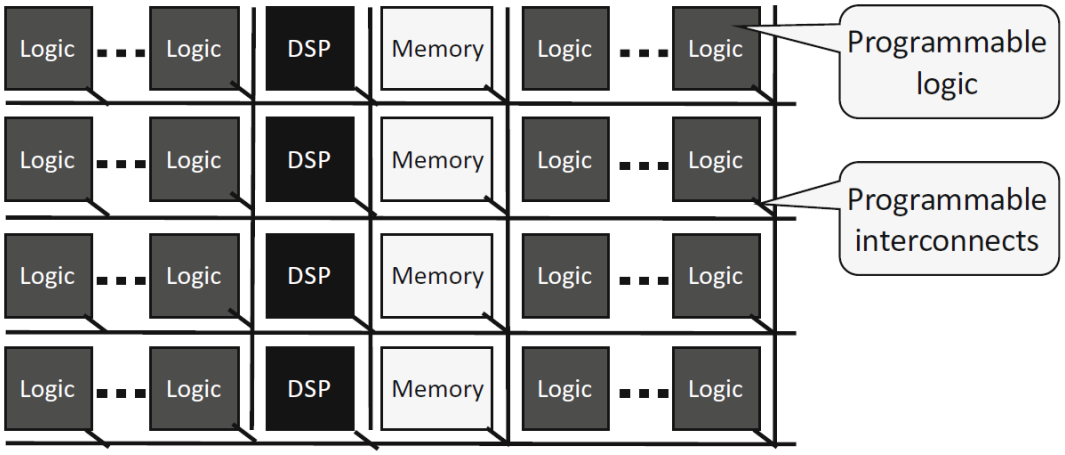


Figure 2.1: FPGA Structure (Waidyasooriya et al., 2018)

FPGA is a type of hardware that is programmable even after it is manufactured. “It contains programmable logic gates and programmable interconnects, as well as configurable memory modules and DSPs” (Waidyasooriya et al., 2018). The programmable logic, interconnects, memory modules and DSPs can be utilized to create any arbitrary circuits. This enables FPGA to “become” different processors and accelerators, for example GPGPU. A processing system is also included in the FPGA. Figure 2.1 shows the structure of FPGA.

2.2. OpenCL Background

"OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPGPUs, DSPs, FPGAs and other processors or hardware accelerators." (OpenCL, 2019). OpenCL consists four models, namely the platform model, execution model, kernel programming model and memory model. Section 2.2.1 provide a brief description of the platform model while Section 2.2.2 explains the execution model. Section 2.2.3 describes the kernel programming model briefly and Section 2.2.4 demonstrates the memory model.

2.2.1. Platform Model

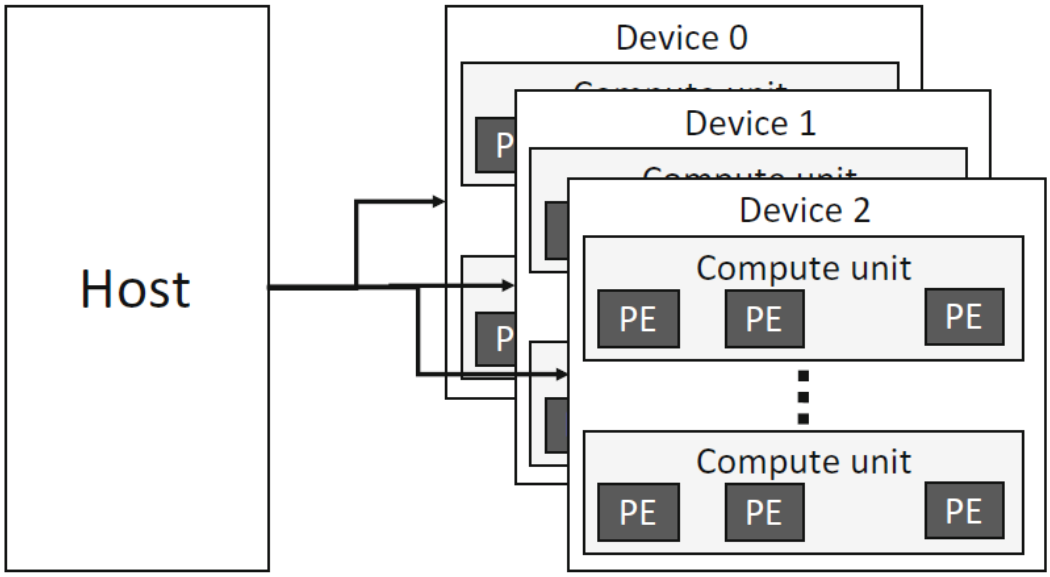


Figure 2.2: OpenCL Platform Model (Waidyasooriya et al., 2018)

As can be seen from Figure 2.2, in a CPU-FPGA heterogeneous system, CPU is the host while FPGAs are the devices. There are multiple compute unit within the devices and there are multiple processing elements (PE) within one single compute unit. PEs and CPU cores are counterparts. Multiple devices can be controlled by one host.

2.2.2. Execution Model

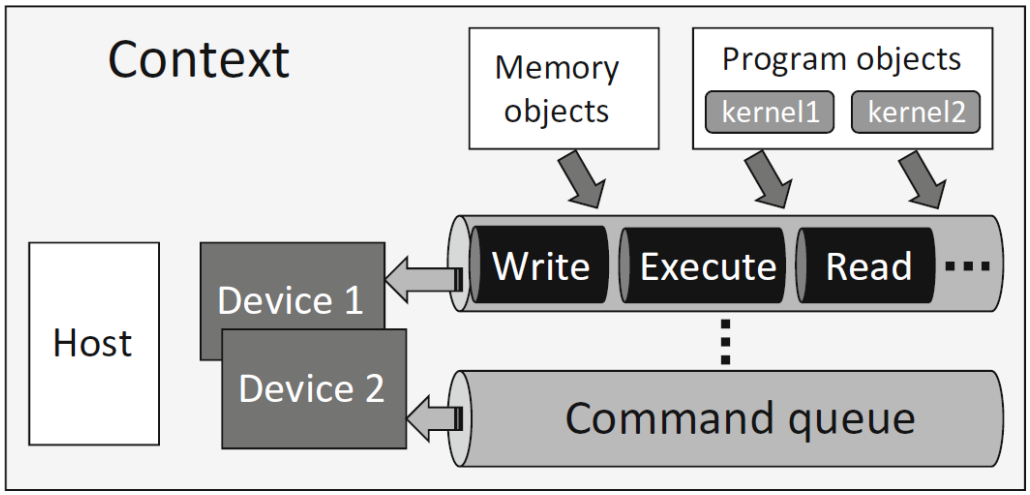


Figure 2.3: OpenCL Execution Model (Waidyasooriya et al., 2018)

Figure 2.3 explains the OpenCL execution model. A context is created for one or multiple devices. Each device can have one or multiple command queues. Command queues are used for communications between host and devices, commands will be issued by host and passed to devices through command queue. There are two types of objects that can be put into the command queue, memory object and program object. Memory objects are the objects that related to the read and write of the memory, while program objects are kernels which needs to be executed. Command queue can be either in-order or out-of-order. Each kernel can be executed by either one or multiple compute units. If the single work item kernel is used, then there is only one work group with one work item in one compute unit. If NDRange kernel is used, then all the following situations are possible depends on the global work item size and the local work item size.

- One work group with multiple work items in one compute unit.
- Multiple work groups each with multiple work items in one compute unit.
- Multiple work groups each with one work item in one compute unit.

2.2.3. Kernel Programming Model

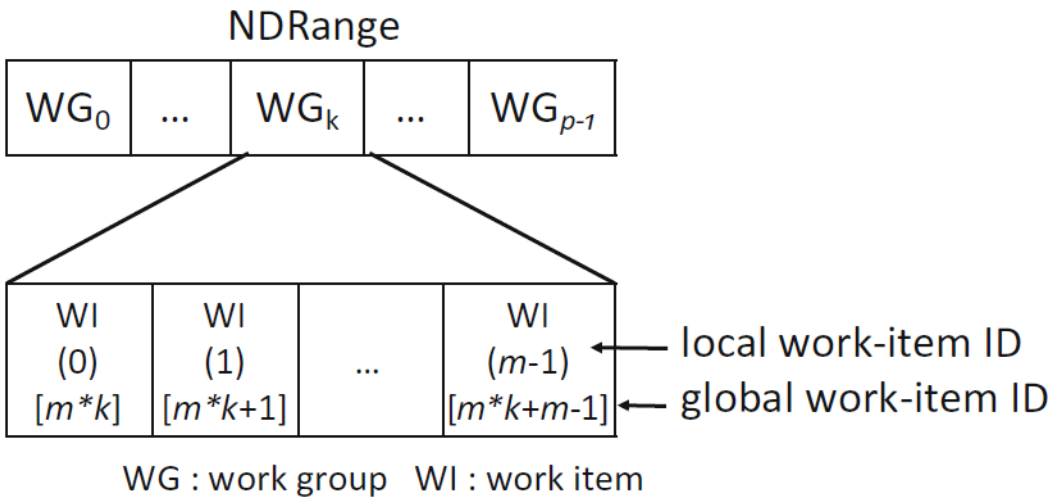


Figure 2.4: OpenCL Kernel Programming Model (Waidyasooriya et al., 2018)

Figure 2.4 demonstrates the OpenCL kernel programming model. Functions executed on device are called kernel. Another new concept is called NDRange kernel. As can be seen from Figure 2.4, a NDRange kernel can be made up of multiple work group while a work group can consist multiple work item. Work groups and work items in a single NDRange kernel can be divided into N dimension, where the maximum number of N is three. “The size of NDRange and work groups can be specified by host program.” (Waidyasooriya et al., 2018). The work items are tagged with both local ID and global ID, the local ID of work items in different work group might be the same but the global ID is unique for different work items. “The local ID is to identify the work items within a work group while global ID is to identify the work items within the NDRange” (Waidyasooriya et al., 2018).

2.2.4. Memory Model

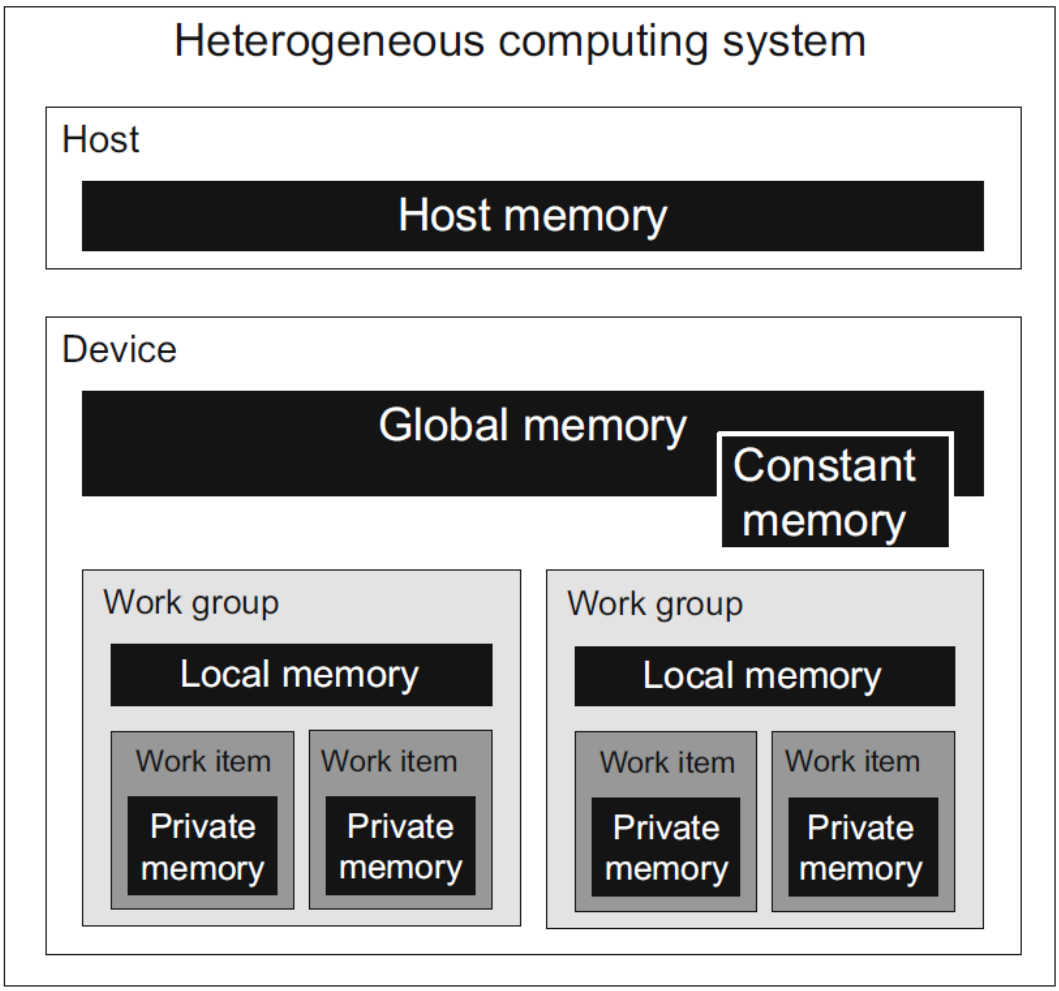


Figure 2.5: OpenCL Memory Model (Waidyasooriya et al., 2018)

The OpenCL memory model is described in Figure 2.5. Host and devices have separate memory, called host memory and device memory respectively. Devices cannot access host memory, so the data in host memory must be transferred to the global memory before it can be processed by device. Global memory can be accessed by both host and device. Constant memory is a read-only memory. Each work group has its own local memory and cannot be accessed by other work groups, but it can be accessed by all the work items within the same work group. Each work item also has its own private memory, which cannot be accessed by other work items. On the FPGA board used in this project, both host memory and global memory refer to the same external and off-chip DDR memory, while local and private memory refer to the on-chip memory.

2.3. Other Programming Models

- OmpSs. OmpSs is a directive-based programming model developed by the Barcelona Supercomputing Centre which aims to extend OpenMP by adding new directives to support asynchronous parallelism and heterogenous computing. “It can also be understood as new directives extending other accelerator-based APIs like CUDA or OpenCL.”. (“The OmpSs Programming Model | Programming Models @ BSC,” n.d.) Asynchronous parallelism is supported in OmpSs by utilizing data dependencies between different tasks of the program. While heterogenous computing is supported by using a newly-introduced construct called the target construct. Architectures supported by OmpSs including Intel 32-bit and 64-bit platforms, IBM Power8 platforms and ARM 32-bit and 64-bit platforms.
- Maxeler high-performance dataflow computing system. Maxeler technology is a company that focus on the domain of high-performance computing. Maxeler high-performance dataflow computing system is one of the software they developed, which consist of MaxIDE, MaxCompiler, MaxOS and MaxGenFD. MaxIDE is an Eclipsed-based IDE, which means the programs execute on the Maxeler system can be developed in Java. The MaxCompiler splits an application into three parts, namely kernel, manager configuration and CPU application, to allow the application utilize the dataflow engine configuration. Kernels implement the computational component of the application. Manager configuration connects kernels to CPU, RAM, other kernels and dataflow engines. CPU application interacts with the dataflow engine to read and write data to kernels and RAM. “MaxelerOS provides the data choreography needed to balance resources, maximize utilization, minimize overheads, and manage the application acceleration process at runtime.”. (“MaxelerOS | Maxeler Technologies,” n.d.) “Maxeler MaxGen systems are domain-specific compilers that enable programmers to easily harness the full power of Maxeler solutions without needing a detailed understanding of the underlying hardware.”. (“MaxGenFD | Maxeler Technologies,” n.d.)
- Vivado HLS. Vivado HLS “accelerates IP creation by enabling C, C++ and System C specifications to be directly targeted into Xilinx programmable devices without the need to manually create RTL” (Vivado High-Level Synthesis, 2019)

- VHDL. “VHDL is an HDL used in electronic design automation to describe digital and mixed-signal systems such as FPGA and IC. VHDL can also be used as a general-purpose parallel programming language.” (VHDL, 2019)

2.4. SDx Background

SDx is an IDE provided by Xilinx for SDSoC and SDAccel development. SDx includes HLS tools like Vivado and is made for Zynq® UltraScale+™ available for this project. SDx has significantly eased the burden of software engineers for programming FPGA. Because Vivado can translate high-level language like C and Java to HDL, a process which is known as high level synthesis. Then the bitstream which builds the hardware architecture of FPGA can be generated by the compiler. The existence of Vivado enables software engineers to program FPGA without using HDL like Verilog, and learning lots of hardware knowledge. Emulators are also provided in SDx to emulate application in a hardware or software environment, in order to verify the functional correctness. It also provides reports regarding performance estimate and hardware resource utilization.

Two reports are provided by SDx, namely the system estimate report and the HLS report. System estimate report provides the estimated clock frequency for execution, hardware resource utilization and latency information of each kernel. It should be noted that the estimated clock frequency doesn’t necessarily equal to the clock frequency set in the SDx project setting page. Hardware resource utilization contains information of the usage of FF, LUT, DSP and BRAM_18K of each kernel. Latency information includes the start interval, best case latency, worst case latency and average case latency of each kernel. Start interval means “the amount of clock cycles that has to pass between invocations of a compute unit for a given kernel.”. While the best, average and worst case latency refer to “how much clock cycles it takes the compute unit to generate the results of one NDRange data tile for the kernel” (“SDAccel Environment Profiling and Optimization Guide,” 2019). The best, worst and average case latency will be the same if there are no dependencies between loop iterations.

One HLS report will be generated for each kernel. The HLS report not only provide the hardware resource utilization and the latency information for the given kernel, it also

provides an analysis of the loops within the kernel. The following information are given in the analysis,

- Latency, which represent the total number of clock cycle that is needed for the whole iteration to complete.
- Iteration latency, which means the clock cycle needed for a single iteration to complete.
- Trip count, which describes the size of the iteration space.
- Initial interval (II), which explains the number of clock cycles needed for a loop iteration to start executing, after the previous iteration starts. It is worth noting that both achieved II and target II are provided here. The target II will also be 1, which is the optimum II.
- The "pipelined" indicates whether the loop is pipelined or not.

2.5. Stencil Computation Background

Stencil computation is widely used in domains like computational fluid dynamics, electromagnetic simulation based on the finite-difference time-domain methods, and iterative solvers of linear equation systems (Sano et al., 2014). Stencil computation means update the value of a certain point on a grid, based on the value of its neighbours. Stencil itself is an area consists of multiple grid point.

There are different methods to compute a stencil, for example Jacobi iteration and Gauss Seidel iteration. Jacobi iteration only needs the values of the grid point from the previous iteration to calculate the new one, which exposes more parallelism by eliminating data dependency. Unlike Jacobi iteration, Gauss Seidel iteration needs both values from the previous and current iteration of the grid point for the computation, which makes it more difficult to be parallelized.

Both Jacobi iteration and Gauss Seidel iteration are so-called "iteration to converge" computation, meaning the result will converge eventually after multiple iterations, and only one grid array is needed for computation. However, there are other kinds of stencil computation which needs multiple grid arrays. For example, "time-stepping" computation like shallow water equation.

Section 2.5.1 provides a description of the common shallow water weather & climate forecasting model, while Section 2.5.2 describes the simplified shallow water application used in this project in details, with the help of the code snippet.

2.5.1. Shallow Water Forecasting Model Background

“The shallow water equations are a set of hyperbolic partial differential equations (or parabolic if viscous shear is considered) that describe the flow below a pressure surface in a fluid (sometimes, but not necessarily, a free surface).” (Shallow water equations, 2019). Shallow water algorithm is a “time-stepping” stencil computation which requires multiple grid arrays. The update of the value at each grid point relies on the values of the previous updated grid array. After all the grid arrays are updated, the first grid array will be updated again, and so on. Shallow water weather & climate forecasting model is an example of sophisticated weather and climate prediction model.

2.5.2. The Simplified Shallow Water Application Used in this Project

```
computation_start = lr_tim();
time = 0.0;
for (ncycle=1;ncycle<=ITMAX;ncycle++)
{
        L100_start = lr_tim();
        kernel_l100 (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu, buffer_h);
        q.finish();
        L100_end = lr_tim();
        time_spent_l100 = time_spent_l100 + (L100_end - L100_start);

        L100_pc_start = lr_tim();
        kernel_l100_pc (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_cu, buffer_h);
        q.finish();
        L100_pc_end = lr_tim();
        time_spent_l100_pc = time_spent_l100_pc + (L100_pc_end - L100_pc_start);

        L200_start = lr_tim();
        kernel_l200 (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu, buffer_h);
        q.finish();
        L200_end = lr_tim();
        time_spent_l200 = time_spent_l200 + (L200_end - L200_start);

        L200_pc_start = lr_tim();
        kernel_l200_pc (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p);
        q.finish();
        L200_pc_end = lr_tim();
        time_spent_l200_pc = time_spent_l200_pc + (L200_pc_end - L200_pc_start);
        time = time + dt;
}
q.finish();
computation_end = lr_tim();
time_spent_computation = time_spent_computation + (computation_end - computation_start);
q.enqueueReadBuffer(buffer_u, CL_TRUE, 0, vector_size_bytes, source_u.data());
q.enqueueReadBuffer(buffer_p, CL_TRUE, 0, vector_size_bytes, source_p.data());
Full_APP_end = lr_tim();
time_spent_Full_app = time_spent_Full_app + (Full_APP_end - Full_APP_start);
```

Figure 2.6: Host Code of Kernel Execution

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100(__global float *u, __global float *p, __global float *cu, __global float *h)
{
        for (i=0;i<M;i++)
        {
                for (j=0;j<N;j++)
                {
                        cu[(i + 1)*M_LEN + (j)] = .33 * (p[(i + 1)*M_LEN + (j)] + p[(i)*M_LEN + (j)] + p[(i)*M_LEN + (j+1)]) + u[(i + 1)*M_LEN + (j)];
                        h[(i)*M_LEN + (j)] =  .16 * (u[(i + 1)*M_LEN + (j)] + p[(i)*M_LEN + (j)]);
                }
        }
        return;
}
```

Figure 2.7: Kernel Code of Kernel L100

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100_pc(__global float *cu, __global float *h)
{
        int i,j;
        for (j=0;j<N;j++)
        {
                cu[(0)*M_LEN + (j)] = cu[(M)*M_LEN + (j)];
                h[(M)*M_LEN + (j)] = h[(0)*M_LEN + (j)];
        }
        for (i=0;i<M;i++)
        {
                cu[(i + 1)*M_LEN + (N)] = cu[(i + 1)*M_LEN + (0)];
                h[(i)*M_LEN + (N)] = h[(i)*M_LEN + (0)];
        }
        cu[(0)*M_LEN + (N)] = cu[(M)*M_LEN + (0)];
        h[(M)*M_LEN + (N)] = h[(0)*M_LEN + (0)];
}
```

Figure 2.8: Kernel Code of Kernel L100_pc

The basic idea behind the simplified shallow water application is quite straight-forward. Figure 2.6 demonstrates how it is implemented. The elements of array cu and array h will first be updated in kernel L100, based on the elements of array u and array p. The periodic continuation operation of array cu and array h will then be conducted in kernel L100_pc. After that, the elements of array u and array p will be updated based on the elements of array cu and array h in kernel L200. Then the periodic continuation operation of array u and array p will be conducted in kernel L200_pc. After all this, the elements of array cu and array h will be updated again. This process will keep iterating until the value of the elements in array u and array p is converged. The update of array cu and array h, as well as the update of array u and array p, can be done in parallel. This is because Jacobi iteration is used here, which means, for example, in iteration 2, the update of array cu and array h only required the array u and array p from iteration 1, which is already available. The situation is similar for the update of array u and array p in iteration 2.

Figure 2.7 and Figure 2.8 provides more details regarding how the update of array cu and array h, as well as their periodic continuation operation are implemented. For example, the update of grid point cu(1, 0) is based on grid point p(1, 0), p(0, 0), p(1, 1) and u(1, 0). The update of grid point h(0, 0) is based on grid point u(1, 0) and p(0, 0). Periodic continuation operation is basically the copy of grid point. In kernel L100_pc, the values of the last row in array cu will be copied to the first row, then the values of the left-most column in array cu will be copied to the right-most column. After that the

grid point cu(65, 0) will be copied to cu(0, 65). In terms of array h, the values of the first row in array h will be copied to the last row, then the values of the right-most column in array h will be copied to the left-most column. After that the grid point h(0, 65) will be copied to h(65, 0). The update as well as the periodic continuation operation of array u and array p is similar with the one of array cu and array h.

Periodic continuation operation leads to "halo", which is an extra circle of data surrounding the original array. Therefore, the problem size is different from the array size. The existence of halo is to provide a better memory access pattern for the system with cache. For example, without halo, if grid point cu(0, 0) needs to be updated, it will require the value from grid point p(0, 0), p(64, 0), p(0, 1) and u(0, 0). It is obvious that the access to grid point p(64, 0) is not a stride-1 access, and the value of grid point p(64, 0) is not likely to be in the cache as well. However, if halo is available, then the update of grid point cu(0, 0) will become the update of grid point cu(1, 0) which is based on grid point p(1, 0), p(0, 0), p(1, 1) and u(1, 0). Although the access to grid point p(0, 0) is still not stride-1 access, the value of grid point p(0, 0) is very likely to be in the cache, which still provides a better memory access pattern compared to the one without halo.

2.6. Optimization Methods for OpenCL Application on Xilinx FPGA

This section summarizes a series of methods, learned from the literature, for optimizing FPGA application developed using OpenCL. The literature from which these methods are derived is summarized in Section 2.7.2. Section 2.6.1 describes the optimizations used in kernel code. Section 2.6.2 demonstrates optimizations used in host code. Section 2.6.3 explains optimizations that are related to the IDE SDx.

2.6.1. Kernel Optimizations

A series of methods that can be used to optimize kernel code are listed as follows,

- On-chip memory. Using on-chip memory as cache can reduce the memory access latency. This means storing data in BRAM, which is also known as local memory in the OpenCL memory model. This can be implemented by declaring the variable as "__local".

- Burst memory transfer. The burst mode can be triggered when copying data between off-chip memory and on-chip memory. Burst memory transfer aims to improve the data transfer efficiency by combining multiple consecutive memory access into one. Hence, the memory bandwidth can be utilized in a more efficient way. This can be implemented by using function async_work_group_copy().
- Loop unrolling. Loop unrolling can improve the parallelism between iterations. It can achieve a better throughput compared to loops that are not unrolled. This can be implemented by using directive "opencl_unroll_hint(n)". The unrolling factor can be specified by changing parameter n. If n is not specified, the loop will be completely unrolled by default. The directive needs to be put ahead of the loop body.
- Loop pipelining. Loop pipelining can improve the parallelism between iterations. According to Fifield et al (Fifield et al., 2016), loop pipelining can achieve the best throughput. This can be implemented by using directive "xcl_pipeline_loop". Section 4.4 provides a detail explanation regarding where to put the "xcl_pipeline_loop" directive in a nested loop.
- Array partitioning. By partitioning the array, the number of logics which can access (read/write) data simultaneously in each clock cycle can be increased. For example, each BRAM block has two data ports, meaning that a maximum of two logics can access data simultaneously in each clock cycle. However, if it is partitioned using directive (cyclic, 2), then each BRAM block will have four data ports, because the array data is distributed to two physical memories. This means that a maximum of four logics can access data simultaneously in each clock cycle.

  This can be implemented using directive "xcl_array_partition(type, factor, dimension)". There are three types of array partitioning method, namely cyclic, block and complete. The type of array partitioning can be specified by changing the "type" parameter. "The original array will be split into equally sized blocks of consecutive elements of the original array, if it is partitioned in a block way. The original array will be split into equally sized blocks interleaving the elements of the original array, if it is partitioned in a cyclic way. The default operation of complete partition is to split the array into its individual elements. This means implementing the array as a collection of registers" ("SDSoC Profiling and

Optimization Guide," 2019). It should be noted that there is an array size threshold of 1024 for complete partition, meaning that arrays with size larger than 1024 cannot be partitioned in a complete way. The factor parameter "can be used to specify the number of arrays which are created for block and cyclic partition. This parameter is not applicable in complete partition. For multi-Dimensional arrays, the dimension option can be used to specify which dimension is partitioned." ("SDSoC Profiling and Optimization Guide," 2019). The directive needs to be placed after the declaration of a variable.

- Data vectorization. Data vectorization can utilize memory bandwidth in a more efficient way by transferring multiple data, instead of one, each clock cycle. This can be implemented either automatically by using directive "vec_type_hint(type)", or manually by declaring variables as, for example, float2, float4, float8 or float16. The type parameter of the "vec_type_hint" directive represent the type of data (double, float and etc) that needs to be vectorized. The directive needs to be placed ahead of the function body.
- The restrict keyword. According to (Zohouri et al., 2016), the restrict keyword can be used so that compiler will avoid making conservative decision like pointer aliasing. Hence, compiler will be able to parallelize loops if there are no dependencies exist. This can be implemented by declaring a pointer as "__restrict".
- Dataflow directive. The dataflow directive can pipeline the execution of the function calls in each kernel. This can be implemented by using directive "xcl_dataflow". The directive needs to be placed ahead of the function body. Figure 2.9 below demonstrates how function calls L100_read(), L100_calc() and L100_write() are pipelined inside kernel L100_cu.

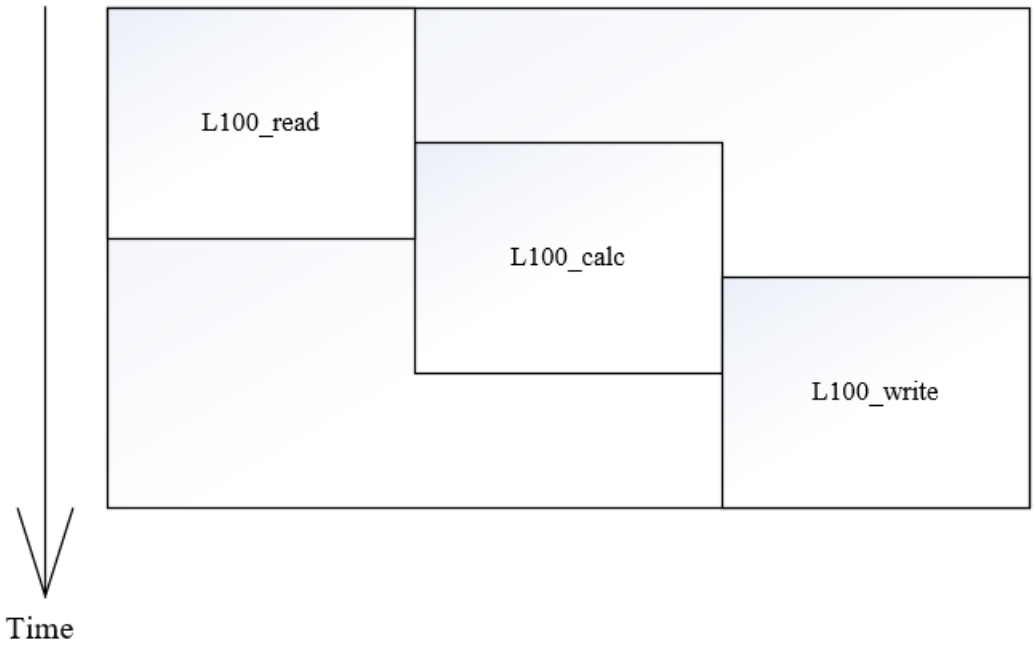


Figure 2.9: Execution of Function Calls L100_read(), L100_calc() and L100_write() after Applying Dataflow Directive

- Function inline. Instead of making function calls, function inline will replace the function call with the function body itself. This may increase the code size, but it also eliminates the time using for function calls, hence improves the performance. This can be implemented by using directive “always_inline”. The directive needs to be placed ahead of the function body.
- Work item pipelining. The work items in a NDRange kernel can be pipelined by using directive “xcl_pipeline_workitems”. It should be noted that this optimization is only available when using NDRange kernel.
- Optimum local work group size specification. The local work group size of a NDRange kernel that yields the best performance will always be the problem size. For example, for problem size of 65 * 65, the local work group size with best performance will be a two-Dimensional work group with size of 65 on each dimension. However, a smaller local work group size will consume fewer hardware resource, which is useful since the hardware resource on a single FPGA is limited. Hence, the optimum work group size should be chosen carefully if applicable. It should be noted that this optimization is only available when using NDRange kernel.
- Pipe. Pipe is a FIFO memory object in the OpenCL programming language, it is very useful when it comes to streaming data between kernels. When the BRAM is not big enough to cache all the data, it will be a good idea to stream data from host

memory to device memory directly without using external memory, in order to minimize data access latency. Pipe can be implemented by using directive "xcl_reqd_pipe_depth(n)", along with function write_pipe_block() and read_pipe_block(). Parameter n defines the size of the pipe. It should be noted that when pipe is utilized to stream data, it is assumed implicitly that the work items are executed sequentially.

- Zero copy of data. "The ZERO_COPY pragma means that the hardware function accesses the data directly from shared memory through an AXI master bus interface." ("SDx Pragma Reference Guide," 2019). This can be done by using pragma "#pragma SDS data zero_copy()". The pragma should be placed ahead of the function body.
- Merging array update kernel with periodic continuation kernel. The array update kernel can be merged with the periodic continuation kernel if the whole array is cached in BRAM, to reduce memory access overhead. For example, kernel L100_cu and kernel L100_pc_cu can be merged. This allows periodic continuation operation to be conducted immediately after array update is complete, to avoid unnecessary data transfer.

2.6.2. Host Optimizations

A series of methods that can be used to optimize host code are listed as follows,

- Overlapping data transfer between host memory and device memory, with the kernel computation. This can be implemented by using function enqueueMigrateMemObjects(). Function enqueueWriteBuffer() and enqueueReadBuffer() need to be replaced by enqueueMigrateMemObjects(). It should be noted that synchronization is needed when using enqueueMigrateMemObjects() in order to obtain the correct result. The parameter "CL_MEM_USE_HOST_PTR" is also be needed when declaring the buffers used by enqueueMigrateMemObjects().
- Concurrent execution of kernels. This can be implemented by using either multiple in-order command queues or one out-of-order command queue. Parameter "CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE" will be necessary when declaring the command queue, if out-of-order command queue is used.
- OpenCL API execution model. Different API should be used to reduce overhead of

kernel enqueuing. “For the data parallel case, use the clEnqueueNDRange API. For the task parallel case, use the clEnqueueTask API.” (“SDAccel Environment Profiling and Optimization Guide,” 2019)

2.6.3. SDx-related Optimizations

A series of methods that are related to SDx are listed as follows,

- The number of compute unit. The number of compute unit can be specified for each kernel in the project setting page. The number of compute unit available for each kernel ranges from 1 to 60.
- The data motion network clock frequency. The data motion network clock frequency can be set in the project setting page. The clock frequency available ranges from 75MHz to 600MHz.
- The port data width. The port data width for each kernel can be set in the project setting page. The width available ranges from 32 bits to 512 bits. The data width can also be set as auto.
- Dedicated memory port for each global array. This can be implemented by ticking the “Max Memory Port” option in the project setting page for each kernel. After this option is selected, each global array within that certain kernel will be assigned with a dedicated memory port.

2.7. Literature Review

This section provides a detailed review to the papers that are key to this project. Papers in Section 2.7.1 mainly focused on performance modelling. Section 2.7.2 reviews the papers that conduct performance optimization to FPGA application. Section 2.7.3 provide a review on a paper that carried out some good overhead analysis. It should be noted in Section 2.7.2, the papers written by Sano et al and Mondigo et al provides both performance optimization and model for performance estimation.

2.7.1. Papers Focused on Performance Modelling

da Silva et al. (da Silva et al., 2013) have proposed a performance model for FPGA by combing the traditional roofline model with the HLS tools. The roofline model provides a performance estimation of the target algorithm by considering the computational

performance (CP), memory bandwidth (BW) and computational intensity (CI). The CP refers to the maximum number of floating-point operations that the processor can achieve. While CI refer to algorithm complexity, which is the number of operations executed per byte access from memory. CP and BW is related to the hardware architecture itself while CI is related to the application.

However, traditional roofline model cannot be applied to estimate the performance of FPGA, because the hardware architecture of FPGA is not fixed and can be influenced by the application. This means CP in this case is directly relates to CI. By connecting the computational power with the resource consumption, an extended roofline model is introduced. A new concept called the scalability (SC) is introduced as well. The SC refer to the number of PE, which can be obtained by dividing the available resources on FPGA by the resources each PE consumes. The performance of a FPGA can then be acquired by multiply SC with the performance of PE. The computation of CI is also modified by dividing the number of operations in one iteration, with the number of memory access for input and output values.

HLS tools play an important role in the extended roofline model. Because they provide optimizations and information like FPGA resource utilization, latency, and throughput. A class of window-based image processing applications along with two different HLS tools are served as case study in their work. They demonstrated that the extended roofline model is accurate enough to estimate the performance of FPGA based on the information provided by the HLS tools. The extended roofline is also flexible enough to be combined with any HLS tools.

Parker (Parker, 2017) has shed some lights on the topic of how to calculate the peak floating-point capabilities of DSP, GPGPU and FPGA in this white paper. One common way of determining the peak FLOPS (floating-point operation per second) rate is to multiply the sum of the adders and multipliers by the maximum operation frequency. Because FLOP is defined as an addition or multiplication of single or double precision number. Other operations like division, square root, FFT (fast Fourier transform) and matrix operation needs to be constructed using adders and multipliers as well.

However, when it comes to FPGA, the method mentioned above might not be able to produce the reliable peak FLOPS rate, due to the following challenges. First, the floating-point precision level implemented on FPGA is not restricted to the industrial-standard single and double precision. In fact, multiple precision levels are implemented. Another challenge is that it is difficult to determine the routing resources that is needed to implement the floating-point operation. While things like large barrel shifter which consumes a large amount of programmable routing are required when implementing floating-point operation. In addition, not all the programmable logic on FPGA can be fully utilized, since it will lead to the reduction of the clock frequency.

In order to calculate the peak FLOPS on FPGA, Parker has proposed two methods. The first one is to use the benchmark provided by the FPGA vendors. Another way is to use only add or subtract function, and build as many adders as possible using DSP48E, then build the remaining adders using pure logic, in order to maximize the floating-point rating. However, he also mentioned that the second method is not a benchmark that is recognized by the industry, and such design has no application benefits. Parker also believed that for FPGA without hard floating-point circuits, using the vendor-calculated theoretical GFLOPS numbers is quite unreliable.

In summary, Parker believed that in order to obtain the peak FLOPS performance of devices with different architectures, for simplicity, it can be done by multiplying the sum of the adders and multipliers by the maximum operation frequency. But ultimately, relevant benchmark provided by the vendors should be used for a more accurate FLOPS performance result.

Some other performance modelling includes a performance modelling of the 3-Dimensional stencil computation on a stream-based FPGA, proposed by (Dohi et al., 2013), a performance modelling of pipelined linear algebra architectures on FPGAs developed by (Skalicky et al., 2013) and a method of evaluating FPGAs for floating-point performance proposed by (Strenski et al., 2008).

2.7.2. Papers Focused on Performance Optimization

Cong et al. (Cong et al., 2018) have proposed a best-effort guideline for improving

FPGA programming productivity as well as FPGA accelerator performance. The guideline makes FPGA programming easier by easing the burden of the software programmers, allowing them to learn fewer hardware knowledge.

Furthermore, the guideline improves the performance of the FPGA accelerator by using five refinement steps of HLS, namely explicit data caching, pipelining, processing element duplication, computation overlapping and scratchpad reorganization. Explicit data caching means explicitly copy the data for the computation to the BRAM of the FPGA. Here BRAM is considered as the cache of FPGA. Pipelining is similar with CPU pipelines. However, FPGA designers can construct very deep pipeline with hundreds or even thousands of stages. Processing element duplication is similar with multithreading programming. The processing units of FPGA and CPU cores is counterparts. Computation overlapping in this case means constructing a three-stage coarse-grained pipeline for better resource utilization. Scratchpad reorganization refers to the using of larger-width data type to better utilize memory bandwidth.

They demonstrated that by applying their guideline, the performance of the FPGA accelerator can be 42~29,030x faster, compared to the performance of the non-optimized, naïve FPGA accelerator. The optimized FPGA accelerator is also up to 112.8x faster than a single Xeon CPU core.

Targett et al. (Targett et al., 2015) have proposed a method of accelerating C-grid shallow water model by using lower precision variable and FPGA. They tried to stop using double precision floating point variable and represent the variable with less bits. By reducing the mantissa length of the variables, the spare computing resources can be used to simulate climate change at a higher resolution, which may eventually improve accuracy.

They also proposed an accuracy verification method, to make sure the accuracy will be acceptable after the precision is lowered. This is accomplished by first calculating the mean and standard deviation of the fields, then the mean and standard deviation of the errors. After that the maximum value of mean and standard deviation is picked out and compared with the acceptable mean and standard deviation.

They demonstrated that the mantissa length of the variable can be reduced to 14 bits while maintaining an acceptable error. Their reduced precision FPGA implementation runs 5.4x faster than the double precision FPGA implementation, and 12x faster than the multi-threaded CPU implementation. What's more, their reduced precision FPGA implementation uses 39 times less energy than the CPU implementation. For the same power consumption, the reduced precision FPGA implementation can compute a 100*100 grid while the CPU implementation can only compute a 29 * 29 grid.

Düben (Düben, 2018) has proposed a method to reduce the overall data usage and data volume by using a new number format which exploits the similarities between ensemble member. The data usage is reduced by reducing the number of bits that represent the information required for the forecasting model. The ensemble mean is removed from the ensemble data and is combined with a normalization by local ensemble range. By doing this the precision is reduced so does the number of bits which represents the number.

The new number format is realized in a standard shallow water model using Fortran. It performs well for long-term, climate-type simulations. However, disadvantages still exist. When utilizing the new number format, ensemble members will be combined into one single simulation which make it impossible to parallelize them. Another disadvantage is that a single simulation with the new number format will take more time compared to the calculation of a single ensemble member. What's more, the total number of floating-point operations is likely to be increased for the entire ensemble forecast.

Sano et al. (Sano et al., 2014) have proposed a custom computing machine (CCM) called scalable streaming-array (SSA) for conducting stencil computation over multiple FPGAs.

The scalable streaming-array is made up of multiple FPGA, with one master FPGA and multiple slave FPGAs. On each FPGA, there is one ISRU (Input Stream Routing Unit), one OSRU (Output Stream Routing Unit) and multiple PSM (Pipelined Stage Model). Each PSM consists of multiple PE, which forms an array connected by a bidirectional

1-Dimensional torus network. The SSA is literally a linear array of PSM, the input values streaming through the ISRU and are assigned to the PEs of the first PSM. Each PE will calculate several stencils base on the value they receive and the value stored in their local buffer. After the calculation is completed, the result will be sent to the corresponding PEs in the next following PSM. The PEs in the same PSM will also needs to communicate with the above and lower PEs to exchange the value they received, since each PE doesn't have all the value it needs to finish the stencil computation. The final output will come from the OSRU which is a single stream.

The scalable streaming-array is designed in this way due to the low operational intensity of stencil computation, as well as the number of iterations needed for allowing the result to converge. Since FPGA is famous for its deep pipeline, each PSM is responsible for the calculation of one iteration. In this way, all the iterations are pipelined, which means multiple iteration computation can be done using a single data stream, and no large memory bandwidth will be needed. Hence, memory access latency is concealed.

A performance model is also proposed for estimating peak performance, scalability and speedup. They demonstrated their scalable streaming-array architecture on multiple high-end and low-end FPGAs. Both 2-Dimensional and 3-Dimensional Jacobi computation are used as benchmark. Their design showed a good agreement with the performance model, and achieved performance of 260 GFlop/s and 236 GFlop/s for 2-Dimensional and 3-Dimensional Jacobi computation, which are 87.4% and 83.9% of the peak performance respectively, with a memory bandwidth of only 2.0 GB/s. In terms of power consumption, the scalable streaming-array architecture provided excellent performance per power of 1.30 GFlop/s/W and 1.07 GFlop/s/W for the 2-Dimensional and 3-Dimensional Jacobi computation respectively. Their design also showed good scalability.

Mondigo et al. (Mondigo et al., 2019) proposed a scalable architecture with deep pipelined stream. What they proposed in this paper is based on the one developed by (Sano et al., 2014). The major contribution of this paper is as follows, it first explained how temporal and spatial parallelism can be achieved, then it presents an inter-FPGA communication subsystem. Finally, it perfected the performance model Sano et al

proposed by considering the inter-FPGA communication overhead, overheads introduced by the temporal parallelism and spatial parallelism.

Temporal parallelism can be achieved by cascading multiple SPE (Streaming Processing Element) to form a deep pipeline. Spatial parallelism can be achieved by having multiple parallelized unit pipelines in each SPE.

The inter-FPGA communication subsystem proposed in this paper is implemented base on a FC (Flow Control) core and a Serial-Lite III (SL3) core. FC core includes TX buffer, RX buffer and credit counter to deal with incoming data stream and backpressure. Both link latency and the depth of the communication buffers will affect the inter-FPGA communication overhead. While the deep pipeline introduced by the temporal parallelism will also lead to overhead. Furthermore, the wider input/output data stream bandwidth introduced by the spatial overhead will cause overhead if either the memory bandwidth or communication link bandwidth is insufficient. By taking all these factors into consideration, the performance model can estimate the theoretical performance accurately. The theoretical performance is different from peak performance by considering the overheads.

They demonstrated their design on multiple cascaded Arria 10 FPGAs using tsunami simulation as benchmark. They found out that the highest scaled performance for 8 cascaded Arria 10 FPGAs is achieved with a single pipeline of 5 SPEs, which obtained a scaled performance of 2.5 TFlops and a parallel efficiency of 98%.

Fifield et al (Fifield et al., 2016) have proposed several methods for optimizing OpenCL application on Xilinx FPGA. In their slide, they first introduced the FPGA architecture and its difference between CPU and GPGPU. Then they talked about the difficulties for programming FPGA.

They proposed several ways of optimizing OpenCL kernels running on Xilinx FPGA, including common optimization like loop unrolling, loop pipelining, work item pipelining, data vectorization, burst memory transfer, array partitioning and the usage of local memory instead of global memory. However, they also proposed some other

unique optimization methods. For example, specifying a better local work group size, using pipes to stream data between kernels and using multiple external memory DDR banks.

Muslim et al (Muslim et al., 2017) has presented an HLS-based FPGA implementation of several algorithms, including KNN (K-Nearest Neighbour) algorithm, Monte Carlo methods for financial models and bitonic sorting algorithm. They also conducted a performance comparison between FPGA and some high-end GPGPU in terms of execution time and power consumption.

The optimizations they used including work item pipelining, loop pipelining, loop unrolling, burst memory transfer, using on-chip memory, using on-chip pipes for inter-kernel communication, using multiple compute units and using dedicated memory port for each global array. Besides, Muslim et al also tried optimizing the algorithm itself. For example, they developed two version of the KNN algorithm. The first version only implemented the distance calculation in the kernel. In the second version however, they implemented both the distance calculation and the nearest neighbour estimation in two different kernels. Muslim et al believed FPGA will perform better for applications that doesn't require too many accesses to slow external DRAM, due to its limited memory bandwidth with it. (compared to GPGPU which has a larger memory bandwidth interface with external DRAM). They also believed that loop pipelining will yield a better performance than loop unrolling since the number of memory port is limited. They claimed that optimizing FPGA application is about guiding the compiler to generate optimized code and memory architecture for each kernel.

They concluded that FPGAs are more energy-efficient than GPGPUs. If FPGA-specific optimizations are applied, FPGA can yield better performance than GPGPU in some test cases as well. For example, when they run Monte Carlo method for Black-Scholes financial model with European vanilla option, The Virtex-7-series FPGA is 2x faster than K4200 GPGPU, and the device power of this FPGA is only 11% of the GTX960 GPGPU. In the case of Black-Scholes model with Asian option, Virtex-7 is at least 2x faster than the GTX960 GPGPU and consumes only 13% of the energy. When they applied the Monte Carlo method on Heston Model with European vanilla option, The

Virtex-7 is 4x faster than the GTX960 GPGPU and uses 7% of the GPGPU energy for this algorithm. In the case of Heston Model with European barrier option, the Virtex-7 FPGA has 5x performance and consumes only 8% of the energy for the same amount of workload, compared to GTX960 GPGPU.

Both Conte (Conte, 2019) and Gorlani (Gorlani, 2017) developed an optimized molecular dynamics application on FPGA. Molecular dynamics computation is a certain kind of stencil computation.

They both introduced optimization methods like using on-chip memory, burst memory transfer, work item pipelining, loop pipelining, loop unrolling, array partitioning, data vectorization and specifying a better local work size.

Gorlani developed two optimized version of molecular dynamics application, namely the “plain version” and the “unroll version”. The plain version is developed based on the baseline version by utilizing optimization methods like work item pipelining, burst memory transfer and on-chip memory. The unroll version is developed based on the plain version by further optimizing it using loop unrolling. Gorlani also introduced the principles of the NDRange kernel. The multiple work items within the NDRange kernel will be executed in a loop (or a nested loop, depends on the dimension of the NDRange kernel), the index of the loop is the magnitude of each dimension of the NDRange kernel. By conducting a series of experiment, Gorlani also managed to prove that in terms of data vectorization, larger vector size will lead to higher data throughput. What’s more, burst memory transfer is fundamental to achieving better performance, the increase of programmable logic clock frequency will also lead to higher data throughput, especially in the case of burst data access.

Conte developed his optimized version of molecular dynamics application by optimizing the neighbour_build kernel and force_compute kernel respectively. The neighbour_build kernel is optimized by using on-chip memory, data vectorization and burst memory transfer, while the force_compute kernel is optimized by using on-chip memory, data vectorization, burst memory transfer and array partitioning. Conte also introduced the relationship between memory port and memory interface and

demonstrated how to improve performance by using multiple memory ports. In fact, the reason why array partitioning might be able to achieve better performance is that by partitioning the array, multiple memory ports will be utilized simultaneously. Hence, the memory bandwidth is used in a more efficient way.

Their optimized molecular dynamics application both achieve improved performance after applying some of the optimization methods, compare to their baseline version. In Conte's molecular dynamics application, the optimized neighbour_build kernel yields a performance improvement of 11% compared with the baseline version, while the optimized force_compute kernel yields a performance improvement of 28% compared with the baseline version.

2.7.3. Paper Focused on Overhead Analysis

Riley et al. (Riley et al., 1997) have proposed a method for developing high-performance parallel program by utilizing a technique called overhead analysis. Molecular dynamic serves as a case study here. The overhead analysis tries to measure, identify and explain all sorts of overhead, then minimize them accordingly. To conduct overhead analysis, a computable solution for the problem needs to be found first, then a candidate implementation needs to be selected and implemented. After that, the execution behaviour needs to be understood. After the overhead analysis is finished, a new candidate implementation or even a new computable solution might be chosen depends on the circumstances.

The overhead analysis is an iterative process and both the performance and cost of the application need to be considered. The process will be stopped if one of the following situations is satisfied, an acceptable performance is achieved, the improvement of the performance is not cost-effective, or efforts run out. A series of overheads are mentioned here including, insufficient parallelism overhead, algorithmic overhead, load imbalance overhead, scheduling overhead, synchronization overhead, remote access overhead and compiler overhead. The situation is summarised using performance curve, including a naïve ideal curve, a realistic ideal curve and an achieved curve.

They demonstrated that by using overhead analysis to understand the observed behaviour of the application, the programmers only need to spend a limited amount of effort to find out the most significant effect that limits the performance. Therefore, the available development resource can be better utilized.

2.8. Summary

The background knowledge needed for this project is presented in detail in this chapter. A general FPGA background is provided, as well as the background of OpenCL language, which includes the platform model, execution model, kernel programming model and the memory model. The IDE used in this project called SDx is discussed. What's more, the stencil computation, which is the computation that shallow water model belongs to, is demonstrated in detail as well. The common shallow water model along with the simplified one is also described. Furthermore, a series of optimization methods for FPGA application developed using OpenCL are presented, which includes optimization for kernel, host and optimization related to SDx. Finally, a literature review is provided including papers focused on performance modelling, performance optimization and overhead analysis.

Chapter 3 Research Methodology

This chapter introduces the research methodology used in this project. Section 3.1 provides a description regarding how performance estimation is conducted in this project. Section 3.2 demonstrates the method for carrying out overhead analysis. Section 3.3 describes a series of optimization method which are applied to the simplified shallow water application, all of these are selected from the methods listed in Section 2.6. Section 3.4 explains how the optimization methods are applied to the simplified shallow water application. Section 3.5 demonstrate a method for efficient FPGA programming. Section 3.6 provide a description about how to obtain the execution time accurately and correctly.

This project implements a simplified shallow water weather & climate forecasting application using C++, OpenCL and an IDE called SDx, provided by Xilinx. An overhead analysis is first carried out towards the baseline code. After that, a series of optimization methods are applied to the baseline code one after another, following a certain principle, which is described in Section 3.4, to tackle the corresponding overhead. A performance estimation is conducted with the most optimized code in each optimization iteration, to evaluate the gap between the estimated performance and the achieved performance. Methods of efficient FPGA programming and accurate execution time acquisition are also developed and applied.

Further details of the experimental set up can be found in Chapter 6.

3.1. Performance Estimation

Performance estimation is to estimate the execution time of a certain algorithm that implemented on FPGA, based on the latency information and clock frequency provided by the system estimate report. The estimated execution time can be obtained using the following formula,

$$T = \sum_{i=1}^{n} \left(\frac{C_i}{f_{est}}\right)$$

where T is the estimated execution time, f_{est} is the estimated clock frequency, C_i is the

latency (in terms of clock cycle) of kernel i. It should be noted that if kernels are executed concurrently, then only the latency of one of the kernels needs to be put into the above formula, assuming there are no load imbalance overheads. However, if load imbalance overheads exist, then the latency of the kernel which requires the longest execution time should be selected.

3.2. Overhead Analysis

The overhead analysis needs to be made to identify and then explain the overheads. The overhead analysis techniques can be referred to the ones proposed by (Riley et al., 1997). Overheads are roughly divided into five categories, namely non-parallel code overhead, load imbalance overhead, scheduling overhead, synchronization overhead and memory access overhead. The existence of sequential code results in non-parallel code overhead. Load imbalance overhead occurs due to the different amount of work possessed by each processor core. Scheduling overhead is the extra code or instruction for scheduling tasks to different processor cores, compared to sequential code. Synchronization overhead is the overhead introduced by synchronization mechanism like lock or barrier. Memory access overhead occurs when data is not in the cache (in this case BRAM) or not in the local memory, so processor must take extra time to fetch data from global memory or remote memory.

3.3. Overhead Minimization

After the overheads are identified and explained, they need to be minimized accordingly. A series of optimization methods that applied to the simplified shallow water application are listed as follows,

In terms of kernel optimization,

- On-chip memory.
- Burst memory transfer.
- Loop unrolling.
- Loop pipelining.
- Array partitioning.
- Data vectorization.

- The restrict keyword.
- Dataflow directive.
- Function inline.

In terms of host optimization,

- Overlapping data transfer between host memory and device memory, with the kernel computation.
- Concurrent execution of kernels.

Other optimization methods including the usage of a better computable solution. For example, kernel L100 can be broke down into kernel L100_cu and L100_h, so the computation of array cu and array h can be done simultaneously, as will be mentioned in Section 5.4. This changing in computable solution will become very useful when it comes to the optimization of concurrent execution of kernels. Furthermore, array update kernel and periodic continuation kernel, for example kernel L100_cu and kernel L100_pc_cu, can be merged to reduce memory access overhead, which will be mentioned in Section 5.6.

The overhead analysis and minimization are iterative processes. These two processes only stop if at least one of the following conditions are met: time or efforts are running out, or the performance is acceptable, or the improvement of the performance is no longer cost-effective.

3.4. The Principle of Applying Optimization Method

A series of optimization methods are applied to the simplified shallow water application one after another. The optimization process is guided by both the execution time and the information obtained from the system estimate report and HLS report, including hardware resource utilization, latency and loop information. The basic idea is, if the application yields a worse performance after applying a certain optimization method, this optimization method will be put aside, and the code will be rolled back to the previous version to try another optimization method. If a significant change in terms of performance fails to be observed after applying a certain optimization method, this optimization method will still be kept.

When it comes to the situation that more than one optimization methods can improve the performance of the application, but only one of them can be chose, then the scalability of the hardware, as well as latency, should be considered. For example, the decision must be made when it comes to using whether loop pipelining or loop unrolling to increase the parallelism between iterations. The hardware resource utilization should be reflected by the performance and the latency. This means a better performance and a lower latency will be expected if more hardware resources are consumed.

3.5. A Method of Efficient FPGA Programming

There are two ways to program the FPGA efficiently, namely relying more on system estimate report and HLS report to guide the optimization process, and using emulator to verify functional correctness. It should be noted that these two methods could be used together.

As mentioned in Section 3.4, the optimization process should be guided by both execution time and the information from the system estimate report and HLS report. However, since it takes at least half an hour to obtain the execution time data, due to the time-consuming hardware compilation (choose actual hardware as compilation target). It would be a better idea if the system estimate report and HLS report is relied more than the actual execution time. The latency information within the system estimate report, which consists the start interval and the best, worst and average case latency, should reflects the performance.

It is worth noting that obtaining the system estimate report and HLS report is much less time-consuming compared to obtaining the execution time. The system estimate report and the HLS report produced by the emulation compilation (choose emulator as the compilation target) is the same as the one produced by the hardware compilation. However, it only takes around five minutes for the emulation compilation to complete. Therefore, in order to program the FPGA efficiently, the code should not be compiled and run on actual FPGA unless significant change of latency information provided by the system estimate report is observed (except for host optimization, because change in host code will not change the system estimate report. The code needs to be compiled and run on FPGA after host optimization is applied).

Although emulator should not be used as a source of execution time or an indicator of performance improvement, it is quite useful when it comes to verify the functional correctness. Because hardware compilation usually takes more than half an hour to finish, it is impractical to verify the functional correctness by executing the code on FPGA. Hence, FPGA should not be used for the verification of functional correctness. However, emulation compilation only takes around five minutes to complete, while starting the emulator is going to take another five minutes. This means it only takes around ten minutes to obtain an output when using emulator. Comparing with the time that is needed for obtaining an output using FPGA, emulator is clearly a better candidate for functional correctness verification.

It should be noticed that the emulator needs to be stopped and restarted each time after modifications are made to the code. This is to make sure the output from the emulator is correct. What's more, it is also recommended to use different SDx project for hardware compilation and emulation compilation. This is because if the code is compiled using emulation compilation first, and hardware compilation later, error will arise when executing the code on FPGA. The reason behind this is not known yet and will be left for future work.

In conclusion, after applying a certain optimization method, the code should be first compiled using emulation compilation, obtain the system estimate report and HLS report then execute it using emulator. If the output is incorrect, then the code should be modified. If the output is correct, then the system estimate report and HLS report should be checked. If there is no significant change in terms of latency, then this optimization will be kept, and another optimization method can be applied to the code. If there is a significant change, then the code should be compiled using hardware compilation and executed on FPGA, to obtain the execution time. A flow chart of a method of efficient FPGA programming is demonstrated as Figure 3.1

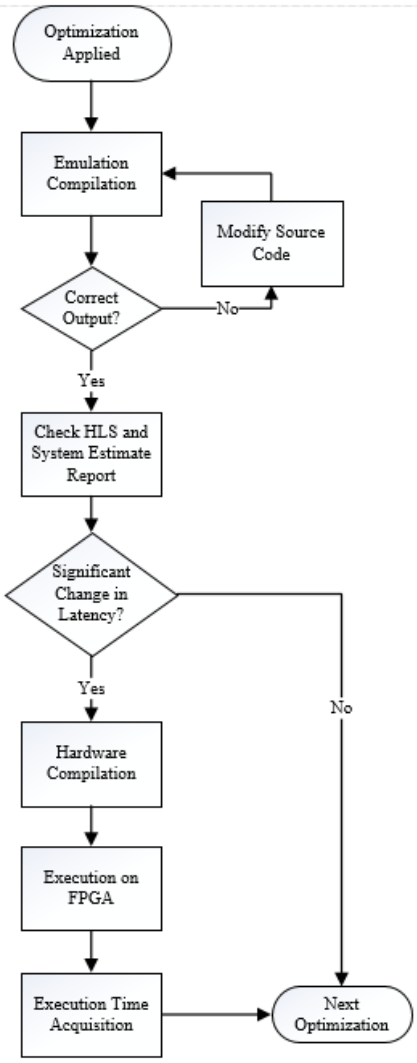


Figure 3.1: Flow Chart of a Method of Efficient FPGA Programming

3.6. Execution Time Acquisition

It should be first noticed that all the execution time measured when executing the code using emulator is not accurate. Therefore, the execution time should only be logged when executing code on FPGA. Furthermore, the emulator should not be used as an indicator of performance improvement either. Sometimes after certain optimization is applied, performance improvement might not able to observe when executing the code using emulator. However, when the exact same code is executed using FPGA, performance improvement (sometimes even significant one) can be observed.

Another worth-noting point is about how to measure the execution time of kernels correctly and accurately. First, the kernel needs to be synchronized every time it finishes

execution. Because in OpenCL kernels are executed asynchronously. Although this will introduce more synchronization overhead, it is the foundation of acquiring accurate timing result. Secondly, the execution time of each kernel needs to be accumulated since the kernels are executed in a loop. Finally, the function used to measure the execution time should be chose carefully. For simplicity the time function of the C library is used instead of the event profiling function provided by OpenCL. However, it should be noted that the gettimeofday() function should be used instead of the clock() and time() function from the time library of C. The problem with time() is that its resolution is not high enough. While the problem with clock() is that, clock() function actually measures the execution time by counting the clock tick of the processor. Considering a heterogenous system of ARM CPU and FPGA is used, and the kernels are executed asynchronously, if the clock() function is utilized, then only the time for the host to call the kernels, instead of the execution time of the kernels, will be measured. This is because the ARM CPU might switch to sleep mode after calling the kernels, so the clock tick will no longer be counted, which leads to inaccurate result produced by the clock() function.

The “time” command of Linux is also used to obtain the real time, user time and system time of the application. By comparing the real time against the execution time measured within the code, the accuracy and correctness of the timing result can be verified.

3.7. Summary

The methodology used in this project is presented in detail in this chapter. The model of performance estimation is discussed, while the overhead analysis is based on a method that classify the overheads into non-parallel code overhead, load imbalance overhead, scheduling overhead, synchronization overhead and memory access overhead. A series of methods including loop unrolling, loop pipelining and array partitioning are used to minimize the overhead. In order to guide the optimization process and program the FPGA efficiently, the dependence on emulator, system estimate report and HLS report is emphasized. Finally, the gettimeofday() function from the C library should be used instead of clock() and time() function to obtain the accurate and correct execution time.

Chapter 4 Experiments with Basic Optimization Methods

This chapter describes the experiments conducted in this project with some basic optimization methods. Section 4.1 demonstrates the baseline code. Section 4.2 explains the reason why emulator cannot be used as the source of execution time as well as the indicator of performance improvement. Section 4.3 describes the using of compiler option of -O3 optimization. Section 4.4 provides a description of optimizing simplified shallow water application by manually pipelining the loop. Section 4.5 demonstrate the optimization of using burst memory transfer and caching data into the on-chip BRAM. Section 4.6 explains the optimization using loop unrolling. The results of the experiments described in this chapter are analysed in Chapter 6.

All the experiments are conducted using problem size of 65 * 65, with an iteration count of 4000 (except for the experiment conducted in Section 4.2), under optimization -O0 (except for the experiment conducted in Section 4.3), using data motion network clock frequency of 99.99 MHz, as well as the single work item kernel of OpenCL. All the experiments are conducted iteratively, meaning that each experiment is based on the previous one. However, it should be noted that the actual array size is 66*66, due to the existence of “halo”, which is discussed in Section 2.5.2. What’s more, the clock frequency that gets set is not necessarily going to be the clock frequency which the hardware executes with.

The reason why no experiments are conducted using NDRange kernel of OpenCL is that, it is believed the performance of the baseline code which uses a single work item kernel, is nearly the same as the one using NDRange kernel with the optimum work group size. When the code is being compiled, the compiler will turn the code in the NDRange kernel into a nested loop, where the iteration space of the loop is the same as the work group size.

```
__attribute__ ((reqd_work_group_size(128, 64, 8)))
__kernel void foobar( ... )
{
  int i = get_local_size(0);
  int j = get_local_size(1);
  int k = get_local_size(2);

  ... // code to be executed

}
```

Figure 4.1: Example Code Snippet of Kernel foobar with a Local Work Group Size of 128 * 64 * 8 (Gorlani, 2017)

```
void foobar( ... )
{
  for(int i = 0; i<128; ++i)
    for(int j = 0; j<64; ++j)
      for(int k = 0; k<8; ++k)
      {
        ... // code to be executed
      }
}
```

Figure 4.2: Kernel foobar being Compiled (Gorlani, 2017)

Figure 4.1 and Figure 4.2 demonstrates what will happen to kernel foobar, which has a local work group size of 128 * 64 * 8, when it is compiled. The code is going to be executed in a nested loop after the kernel is compiled. The iteration space of each layer of the loop is the same as the magnitude of each dimension of the local work group. For example, the magnitude of x-dimension of the local work group is 128, so the iteration space of the outer loop is 128 as well. The situations are the same between the magnitude of y-dimension of the local work group and the iteration space of the medium loop, as well as the magnitude of z-dimension of the local work group and the iteration space of the inner loop.

4.1. Baseline

This experiment aims to turn a simplified shallow water application, which is developed using C++ and OpenCL, and meant to be executed on GPGPU, into a version that can

be executed on Xilinx FPGA. The execution time, as well as the hardware resource utilization, latency and loop information are logged to serve as a baseline.

```
std::vector<cl::Device> devices = xcl::get_xil_devices();
cl::Device device = devices[0];
std::string device_name = device.getInfo<CL_DEVICE_NAME>();

cl::Context context(device);
cl::CommandQueue q(context, device);

std::string binaryFile = xcl::find_binary_file(device_name,"simple_shallow_kernel");
cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
devices.resize(1);
cl::Program program(context, devices, bins);

cl::Kernel kernel1(program,"l100", &err);
auto kernel_l100 = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel1);
cl::Kernel kernel2(program,"l200", &err);
auto kernel_l200 = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel2);
cl::Kernel kernel3(program,"l100_pc", &err);
auto kernel_l100_pc = cl::KernelFunctor<cl::Buffer&, cl::Buffer&>(kernel3);
cl::Kernel kernel4(program,"l200_pc", &err);
auto kernel_l200_pc = cl::KernelFunctor<cl::Buffer&, cl::Buffer&>(kernel4);

cl::Buffer buffer_p (context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_u (context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_cu (context, CL_MEM_READ_WRITE,vector_size_bytes);
cl::Buffer buffer_h (context, CL_MEM_READ_WRITE,vector_size_bytes);

q.enqueueWriteBuffer(buffer_p, CL_TRUE, 0, vector_size_bytes, source_p.data());
q.enqueueWriteBuffer(buffer_u, CL_TRUE, 0, vector_size_bytes, source_u.data());
```

Figure 4.3: Baseline Host Code of the Initialization of Command Queue, Buffers and Kernels, plus Data Copy

```
computation_start = lr_tim();
time = 0.0;
for (ncycle=1;ncycle<=ITMAX;ncycle++)
{
        L100_start = lr_tim();
        kernel_l100 (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu, buffer_h);
        q.finish();
        L100_end = lr_tim();
        time_spent_l100 = time_spent_l100 + (L100_end - L100_start);

        L100_pc_start = lr_tim();
        kernel_l100_pc (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_cu, buffer_h);
        q.finish();
        L100_pc_end = lr_tim();
        time_spent_l100_pc = time_spent_l100_pc + (L100_pc_end - L100_pc_start);

        L200_start = lr_tim();
        kernel_l200 (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu, buffer_h);
        q.finish();
        L200_end = lr_tim();
        time_spent_l200 = time_spent_l200 + (L200_end - L200_start);

        L200_pc_start = lr_tim();
        kernel_l200_pc (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p);
        q.finish();
        L200_pc_end = lr_tim();
        time_spent_l200_pc = time_spent_l200_pc + (L200_pc_end - L200_pc_start);
        time = time + dt;
}
q.finish();
computation_end = lr_tim();
time_spent_computation = time_spent_computation + (computation_end - computation_start);
q.enqueueReadBuffer(buffer_u, CL_TRUE, 0, vector_size_bytes, source_u.data());
q.enqueueReadBuffer(buffer_p, CL_TRUE, 0, vector_size_bytes, source_p.data());
Full_APP_end = lr_tim();
time_spent_Full_app = time_spent_Full_app + (Full_APP_end - Full_APP_start);
```

Figure 4.4: Baseline Host Code of Kernel Execution and the Copy-Back of Data

```
double lr_tim()
{
   timeval tim;
   gettimeofday(&tim,NULL);
   return ( (double)tim.tv_sec + (double)tim.tv_usec/1000000 );
}
```

Figure 4.5: Timing Function based on gettimeofday()

Figure 4.3 and Figure 4.4 demonstrate the host C++ code of the simplified shallow water application. The host code includes getting the Xilinx platform and device, loading the XCL binary file, as well as the declaration of an in-order command queue, kernels and buffers. After that, the data needed for computation will be transferred from the host memory to the device's global memory. The execution of the kernels will get started once the data transfer is completed. The kernels will be synchronized every time after they finish execution, so the execution time of each kernel can be measured. After all the kernels finish execution, the data will be copied back to host memory. Figure 4.5 shows the timing function based on the gettimeofday() function from the C library.

4.2. The Unreliable Emulator

This experiment is to demonstrate that emulator cannot be used as the source of execution time as well as the indicator of performance improvement. The baseline code is executed on both emulator and FPGA with an iteration count of 2. Then the execution time from both emulator and FPGA are recorded respectively and compared against each other. The reason why the simplified shallow water application only executes with 2 iterations is that, the execution time of the code running on emulator is significantly longer than the execution time of the code running on FPGA. If the simplified shallow water application executes with 4000 iterations on emulator, the execution time will be so long that it becomes impractical. An optimized version of simplified shallow water application (from Section 4.5) is also executed on both emulator and FPGA, the execution time obtained from both emulator and FPGA are logged respectively and compared against each other.

4.3. *Iteration 1*: -O3 Optimization

This experiment is investigating the option of optimizing the simplified shallow water application using one of the easiest ways, the -O3 optimization option provided by the

compiler. This experiment builds on the baseline experiment conducted in Section 4.1.

4.4. *Iteration 2*: Loop Pipelining

This experiment is investigating the option of manually pipelining the loops in kernel L100_pc, L200 and L200_pc by using the "xcl_pipeline_loop" directive as mentioned in Section 2.6.1.

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l200(__global float *u, __global float *p, __global float *cu, __global float *h)
{
	for (i=0;i<M;i++)
	{
		__attribute__((xcl_pipeline_loop))
		for (j=0;j<N;j++)
		{
			u[(i + 1)*M_LEN + (j)] = 0.16 * (cu[(i + 1)*M_LEN + (j)] + h[(i)*M_LEN + (j)]);
			p[(i)*M_LEN + (j)] = 0.33 * (cu[(i + 1)*M_LEN + (j)] + cu[(i)*M_LEN + (j)] + cu[(i+1)*M_LEN + (j + 1)]) - h[(i)*M_LEN + (j)];
		}
	}
	return;
}
```

Figure 4.6: Kernel Code of Loops in Kernel L200 Being Manually Pipelined

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l200_pc(__global float *u, __global float *p)
{
	int i,j;
	__attribute__((xcl_pipeline_loop))
	for (j=0;j<N;j++)
	{
		u[(0)*M_LEN + (j)] = u[(M)*M_LEN + (j)];
		p[(M)*M_LEN + (j)] = p[(0)*M_LEN + (j)];
	}
	__attribute__((xcl_pipeline_loop))
	for (i=0;i<M;i++)
	{
		u[(i + 1)*M_LEN + (N)] = u[(i + 1)*M_LEN + (0)];
		p[(i)*M_LEN + (N)] = p[(i)*M_LEN + (0)];
	}
	u[(0)*M_LEN + (N)] = u[(M)*M_LEN + (0)];
	p[(M)*M_LEN + (N)] = p[(0)*M_LEN + (0)];
}
```

Figure 4.7: Kernel Code of Loops in Kernel L200_pc Being Manually Pipelined

Figure 4.6 and Figure 4.7 shows how the "xcl_pipeline_loop" directive is used to manually pipeline the loops in kernel L200 and L200_pc. The situation in kernel L100_pc is similar with the one in L200_pc. The reason why the "xcl_pipeline_loop" directive is put in the inner loop of kernel L200 is that, "the pipeline optimization directive should be placed at the level where a sample of data is processed. Data samples—a frame of data—typically supplied as an array or pointer with data accessed through pointer arithmetic during each transaction" ("SDSoC Profiling and Optimization Guide," 2019). This experiment builds on the baseline experiment

conducted in Section 4.1.

4.5. *Iteration 3*: Using Local Memory and Burst Memory Transfer

This experiment is investigating the option of using function async_work_group_copy() to trigger burst memory transfer. What's more, the data will be stored in BRAM by declaring the array within the kernel function as "__local". Both methods are mentioned in Section 2.6.1.

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100(__global float *u, __global float *p, __global float *cu, __global float *h)
{
	__local float u_buff[wg_size];
	__local float p_buff[wg_size];
	__local float cu_buff[wg_size];
	__local float h_buff[wg_size];
	async_work_group_copy(u_buff, u, wg_size, 0);
	async_work_group_copy(p_buff, p, wg_size, 0);
	async_work_group_copy(cu_buff, cu, wg_size, 0);
	async_work_group_copy(h_buff, h, wg_size, 0);
	for (i=0;i<M;i++)
	{
		for (j=0;j<N;j++)
		{
			cu_buff[(i + 1)*M_LEN + (j)] = .33 * (p_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j+1)]) + u_buff[(i + 1)*M_LEN + (j)];
			h_buff[(i)*M_LEN + (j)] =  .16 * (u_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)]);
		}
	}
	async_work_group_copy(u, u_buff, wg_size, 0);
	async_work_group_copy(p, p_buff, wg_size, 0);
	async_work_group_copy(cu, cu_buff, wg_size, 0);
	async_work_group_copy(h, h_buff, wg_size, 0);
	return;
}
```

Figure 4.8: Kernel Code of Local Memory and Burst Memory Transfer Being Used in Kernel L100

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100_pc(__global float *cu, __global float *h)
{
	__local float cu_buff[wg_size];
	__local float h_buff[wg_size];
	async_work_group_copy(cu_buff, cu, wg_size, 0);
	async_work_group_copy(h_buff, h, wg_size, 0);
	int i,j;
	for (j=0;j<N;j++)
	{
		cu_buff[(0)*M_LEN + (j)] = cu_buff[(M)*M_LEN + (j)];
		h_buff[(M)*M_LEN + (j)] = h_buff[(0)*M_LEN + (j)];
	}
	for (i=0;i<M;i++)
	{
		cu_buff[(i + 1)*M_LEN + (N)] = cu_buff[(i + 1)*M_LEN + (0)];
		h_buff[(i)*M_LEN + (N)] = h_buff[(i)*M_LEN + (0)];
	}
	cu_buff[(0)*M_LEN + (N)] = cu_buff[(M)*M_LEN + (0)];
	h_buff[(M)*M_LEN + (N)] = h_buff[(0)*M_LEN + (0)];
	async_work_group_copy(cu, cu_buff, wg_size, 0);
	async_work_group_copy(h, h_buff, wg_size, 0);
}
```

Figure 4.9: Kernel Code of Local Memory and Burst Memory Transfer Being Used in Kernel L100_pc

Figure 4.8 and Figure 4.9 demonstrate how this is implemented in kernel L100 and L100_pc, the situation in kernel L200 and L200_pc is similar. In Figure 4.8, the first four async_work_group_copy() statements are defined as "load" operation, while the last four async_work_group_copy() statements are defined as "store" operation. The

nested loop between the async_work_group_copy() statements are defined as “calc” operation. This experiment builds on the baseline experiment conducted in Section 4.1.

4.6. *Iteration 4*: Loop Unrolling

This experiment is investigating the option of unrolling the loops in kernel L100, L100_pc, L200 and L200_pc, using “opencl_unroll_hint” directive as mentioned in Section 2.6.1. This experiment aims to compare the different effect posed on performance when using loop pipelining and loop unrolling.

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100(__global float *u, __global float *p, __global float *cu, __global float *h)
{
	__local float u_buff[wg_size];
	__local float p_buff[wg_size];
	__local float cu_buff[wg_size];
	__local float h_buff[wg_size];
	async_work_group_copy(u_buff, u, wg_size, 0);
	async_work_group_copy(p_buff, p, wg_size, 0);
	async_work_group_copy(cu_buff, cu, wg_size, 0);
	async_work_group_copy(h_buff, h, wg_size, 0);
	for (i=0;i<M;i++)
	{
		__attribute__((opencl_unroll_hint))
		for (j=0;j<N;j++)
		{
			cu_buff[(i + 1)*M_LEN + (j)] = .33 * (p_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j+1)]) + u_buff[(i + 1)*M_LEN + (j)];
			h_buff[(i)*M_LEN + (j)] =  .16 * (u_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)]);
		}
	}
	async_work_group_copy(u, u_buff, wg_size, 0);
	async_work_group_copy(p, p_buff, wg_size, 0);
	async_work_group_copy(cu, cu_buff, wg_size, 0);
	async_work_group_copy(h, h_buff, wg_size, 0);
	return;
}
```

Figure 4.10: Kernel Code of the Inner Loop in Kernel L100 Being Completely Unrolled

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100_pc(__global float *cu, __global float *h)
{
	__local float cu_buff[wg_size];
	__local float h_buff[wg_size];
	async_work_group_copy(cu_buff, cu, wg_size, 0);
	async_work_group_copy(h_buff, h, wg_size, 0);
	int i,j;
	__attribute__((opencl_unroll_hint))
	for (j=0;j<N;j++)
	{
		cu_buff[(0)*M_LEN + (j)] = cu_buff[(M)*M_LEN + (j)];
		h_buff[(M)*M_LEN + (j)] = h_buff[(0)*M_LEN + (j)];
	}
	__attribute__((opencl_unroll_hint))
	for (i=0;i<M;i++)
	{
		cu_buff[(i + 1)*M_LEN + (N)] = cu_buff[(i + 1)*M_LEN + (0)];
		h_buff[(i)*M_LEN + (N)] = h_buff[(i)*M_LEN + (0)];
	}
	cu_buff[(0)*M_LEN + (N)] = cu_buff[(M)*M_LEN + (0)];
	h_buff[(M)*M_LEN + (N)] = h_buff[(0)*M_LEN + (0)];
	async_work_group_copy(cu, cu_buff, wg_size, 0);
	async_work_group_copy(h, h_buff, wg_size, 0);
}
```

Figure 4.11: Kernel Code of the Loops in Kernel L100_pc Being Completely Unrolled

Figure 4.10 and Figure 4.11 demonstrates how this is implemented in kernel L100 and L100_pc. The situation in kernel L200 and L200_pc is similar. An experiment of unrolling the loops in kernels L100, L100_pc, L200 and L200_pc using a factor of 8 is

also conducted. It can be achieved by simply replacing directive “opencl_unroll_hint” with “opencl_unroll_hint(8)”. This experiment is based on the experiment conducted in Section 4.5.

4.7. Summary

The experiments described in this chapter are summarized in the following table,

Optimization	-O3 Optimization	Loop Pipelining	Local Memory and Burst Memory Transfer	Loop Unrolling
Baseline	X	X	X	X
Iteration 1	√	X	X	X
Iteration 2	X	√	X	X
Iteration 3	X	√ (auto*)	√	X
Iteration 4	X	X	√	√

*: auto means the loops are pipelined automatically by the compiler after optimizations local memory and burst memory transfer are applied.

Table 4.1: Summary of Experiments Described in Chapter 4

Chapter 5 Experiment with Advanced Optimization Methods

This chapter describes the experiments conducted in this project with some more advanced optimization methods. Section 5.1 demonstrates the optimization using array partitioning. Section 5.2 explains how to optimize the code using automatic data vectorization. Section 5.3 describes the optimization of overlapping data transfer with kernel computation. Section 5.4 provide a description of optimizing simplified shallow water application by using the restrict keyword and the concurrent execution of kernels. Section 5.5 demonstrate the optimization of using function calls pipelining and function inline. Section 5.6 provides a description of merging array update operation with periodic continuation operation. The results of the experiments described in this chapter are analysed in Chapter 6.

5.1. *Iteration 5*: Array Partitioning

This experiment is investigating the option of partitioning the array using "xcl_array_partition()" directive as mentioned in Section 2.6.1. Array partitioning can solve the "limited memory port" warning message arise during the compilation.

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100(__global float *u, __global float *p, __global float *cu, __global float *h)
{
        __local float u_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float p_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float cu_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float h_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        async_work_group_copy(u_buff, u, wg_size, 0);
        async_work_group_copy(p_buff, p, wg_size, 0);
        async_work_group_copy(cu_buff, cu, wg_size, 0);
        async_work_group_copy(h_buff, h, wg_size, 0);
        for (i=0;i<M;i++)
        {
                for (j=0;j<N;j++)
                {
                        cu_buff[(i + 1)*M_LEN + (j)] = .33 * (p_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j+1)]) + u_buff[(i + 1)*M_LEN + (j)];
                        h_buff[(i)*M_LEN + (j)] =  .16 * (u_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)]);
                }
        }
        async_work_group_copy(u, u_buff, wg_size, 0);
        async_work_group_copy(p, p_buff, wg_size, 0);
        async_work_group_copy(cu, cu_buff, wg_size, 0);
        async_work_group_copy(h, h_buff, wg_size, 0);
        return;
}
```

Figure 5.1: Kernel Code of the Local Arrays in Kernel L100 Being Partitioned

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100_pc(__global float *cu, __global float *h)
{
        __local float cu_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float h_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        async_work_group_copy(cu_buff, cu, wg_size, 0);
        async_work_group_copy(h_buff, h, wg_size, 0);
        int i,j;
        for (j=0;j<N;j++)
        {
                cu_buff[(0)*M_LEN + (j)] = cu_buff[(M)*M_LEN + (j)];
                h_buff[(M)*M_LEN + (j)] = h_buff[(0)*M_LEN + (j)];
        }
        for (i=0;i<M;i++)
        {
                cu_buff[(i + 1)*M_LEN + (N)] = cu_buff[(i + 1)*M_LEN + (0)];
                h_buff[(i)*M_LEN + (N)] = h_buff[(i)*M_LEN + (0)];
        }
        cu_buff[(0)*M_LEN + (N)] = cu_buff[(M)*M_LEN + (0)];
        h_buff[(M)*M_LEN + (N)] = h_buff[(0)*M_LEN + (0)];
        async_work_group_copy(cu, cu_buff, wg_size, 0);
        async_work_group_copy(h, h_buff, wg_size, 0);
}
```

Figure 5.2: Kernel Code of the Local Arrays in Kernel L100_pc Being Partitioned

Figure 5.1 and Figure 5.2 show how this is implemented in kernel L100 and L100_pc. The situation in L200 and L200_pc is similar. The loops in kernel L100_pc and L200_pc is unrolled using a factor of 2 because it can minimize the initial interval to 1. This experiment is based on the experiment conducted in Section 4.5.

5.2. *Iteration 6*: Data Vectorization

This experiment is investigating the option of vectorizing the array elements automatically by using “vec_type_hint” directive as mentioned in Section 2.6.1.

```
__attribute__ ((vec_type_hint(float)))
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100(__global float *u, __global float *p, __global float *cu, __global float *h)
{
        __local float u_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float p_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float cu_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float h_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        async_work_group_copy(u_buff, u, wg_size, 0);
        async_work_group_copy(p_buff, p, wg_size, 0);
        async_work_group_copy(cu_buff, cu, wg_size, 0);
        async_work_group_copy(h_buff, h, wg_size, 0);
        for (i=0;i<M;i++)
        {
                for (j=0;j<N;j++)
                {
                        cu_buff[(i + 1)*M_LEN + (j)] = .33 * (p_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j+1)]) + u_buff[(i + 1)*M_LEN + (j)];
                        h_buff[(i)*M_LEN + (j)] =  .16 * (u_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)]);
                }
        }
        async_work_group_copy(u, u_buff, wg_size, 0);
        async_work_group_copy(p, p_buff, wg_size, 0);
        async_work_group_copy(cu, cu_buff, wg_size, 0);
        async_work_group_copy(h, h_buff, wg_size, 0);
        return;
}
```

Figure 5.3: Kernel Code of the Local Arrays in Kernel L100 Being Vectorized Automatically

```
__attribute__ ((vec_type_hint(float)))
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100_pc(__global float *cu, __global float *h)
{
        __local float cu_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float h_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        async_work_group_copy(cu_buff, cu, wg_size, 0);
        async_work_group_copy(h_buff, h, wg_size, 0);
        int i,j;
        for (j=0;j<N;j++)
        {
                cu_buff[(0)*M_LEN + (j)] = cu_buff[(M)*M_LEN + (j)];
                h_buff[(M)*M_LEN + (j)] = h_buff[(0)*M_LEN + (j)];
        }
        for (i=0;i<M;i++)
        {
                cu_buff[(i + 1)*M_LEN + (N)] = cu_buff[(i + 1)*M_LEN + (0)];
                h_buff[(i)*M_LEN + (N)] = h_buff[(i)*M_LEN + (0)];
        }
        cu_buff[(0)*M_LEN + (N)] = cu_buff[(M)*M_LEN + (0)];
        h_buff[(M)*M_LEN + (N)] = h_buff[(0)*M_LEN + (0)];
        async_work_group_copy(cu, cu_buff, wg_size, 0);
        async_work_group_copy(h, h_buff, wg_size, 0);
}
```

Figure 5.4: Kernel Code of the Local Arrays in Kernel L100_pc Being Vectorized Automatically

Figure 5.3 and Figure 5.4 shows how this can be implemented in kernel L100 and L100_pc. The situation in kernel L200 and L200_pc is similar. This experiment is based on the experiment conducted in Section 5.1.

5.3. *Iteration 7*: Overlapping Data Transfer with Kernel Computation

This experiment is investigating the option of overlapping the data transfer between host and device along with the kernel computation, by using function enqueueMigrateMemObjects() as mentioned in Section 2.6.2.

```
cl::Buffer buffer_p (context, CL_MEM_USE_HOST_PTR, vector_size_bytes, source_p.data());
cl::Buffer buffer_u (context, CL_MEM_USE_HOST_PTR, vector_size_bytes, source_u.data());
cl::Buffer buffer_cu (context, CL_MEM_READ_WRITE,vector_size_bytes);
cl::Buffer buffer_h (context, CL_MEM_READ_WRITE,vector_size_bytes);

q.enqueueMigrateMemObjects({buffer_p, buffer_u},0);
q.finish();
computation_start = lr_tim();
time = 0.0;
for (ncycle=1;ncycle<=ITMAX;ncycle++)
{
        L100_start = lr_tim();
        kernel_l100 (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu, buffer_h);
        q.finish();
        L100_end = lr_tim();
        time_spent_l100 = time_spent_l100 + (L100_end - L100_start);

        L100_pc_start = lr_tim();
        kernel_l100_pc (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_cu, buffer_h);
        q.finish();
        L100_pc_end = lr_tim();
        time_spent_l100_pc = time_spent_l100_pc + (L100_pc_end - L100_pc_start);

        L200_start = lr_tim();
        kernel_l200 (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu, buffer_h);
        q.finish();
        L200_end = lr_tim();
        time_spent_l200 = time_spent_l200 + (L200_end - L200_start);

        L200_pc_start = lr_tim();
        kernel_l200_pc (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p);
        q.finish();
        L200_pc_end = lr_tim();
        time_spent_l200_pc = time_spent_l200_pc + (L200_pc_end - L200_pc_start);
        time = time + dt;
}
q.finish();
computation_end = lr_tim();
time_spent_computation = time_spent_computation + (computation_end - computation_start);
q.enqueueMigrateMemObjects({buffer_p, buffer_u},CL_MIGRATE_MEM_OBJECT_HOST);
q.finish();
Full_APP_end = lr_tim();
time_spent_Full_app = time_spent_Full_app + (Full_APP_end - Full_APP_start);
```

Figure 5.5: Host Code of Using enqueueMigrateMemObjects() to Overlap Data Transfer with Kernel Computation

```
__attribute__ ((vec_type_hint(float)))
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100(__global float *u, __global float *p, __global float *cu, __global float *h)
{
        __local float u_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float p_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float cu_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float h_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        async_work_group_copy(cu_buff, cu, wg_size, 0);
        async_work_group_copy(h_buff, h, wg_size, 0);
        for (i=0;i<M;i++)
        {
                for (j=0;j<N;j++)
                {
                        cu_buff[(i + 1)*M_LEN + (j)] = .33 * (p_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j+1)]) + u_buff[(i + 1)*M_LEN + (j)];
                        h_buff[(i)*M_LEN + (j)] =  .16 * (u_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)]);
                }
        }
        async_work_group_copy(cu, cu_buff, wg_size, 0);
        async_work_group_copy(h, h_buff, wg_size, 0);
        return;
}
```

Figure 5.6: Kernel Code of Kernel L100 Being Optimized by Using Fewer async_work_group_copy()

Figure 5.5 demonstrates how this is implemented in host code. The enqueueWriteBuffer() and enqueueReadBuffer() function needs to be replaced with enqueueMigrateMemObjects(), with different parameter. The parameter "CL_MEM_USE_HOST_PTR" needs to be added to the declaration of buffer_p and buffer_u as well. It is worth noting that the enqueueMigrateMemObjects() function needs to be synchronized to obtain the correct result. Figure 5.6 demonstrates another optimization by using fewer async_work_group_copy() function. This means only the data that is necessary for computation will be copied in and out between the on-chip and off-chip memory. This experiment is based on the experiment conducted in Section 5.2.

5.4. *Iteration 8*: Restrict Keyword and Concurrent Execution of Kernels

This experiment is investigating the option of using multiple in-order command queues or one out-of-order command queue to execute the kernels concurrently. What's more, the "__restrict" keyword is also utilized for compiler optimization. The optimization methods of "__restrict" keyword and concurrent execution of kernels are mentioned in Section 2.6.1 and Section 2.6.2 respectively.

```
std::vector<cl::Device> devices = xcl::get_xil_devices();
cl::Device device = devices[0];
std::string device_name = device.getInfo<CL_DEVICE_NAME>();

cl::Context context(device);
cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE | CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);

std::string binaryFile = xcl::find_binary_file(device_name,"simple_shallow_kernel");
cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
devices.resize(1);
cl::Program program(context, devices, bins);

cl::Kernel kernel1(program,"l100_cu", &err);
auto kernel_l100_cu = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel1);
cl::Kernel kernel2(program,"l100_h", &err);
auto kernel_l100_h = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel2);
cl::Kernel kernel3(program,"l200_u", &err);
auto kernel_l200_u = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel3);
cl::Kernel kernel4(program,"l200_p", &err);
auto kernel_l200_p = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel4);
cl::Kernel kernel5(program,"l100_pc", &err);
auto kernel_l100_pc = cl::KernelFunctor<cl::Buffer&, cl::Buffer&>(kernel5);
cl::Kernel kernel6(program,"l200_pc", &err);
auto kernel_l200_pc = cl::KernelFunctor<cl::Buffer&, cl::Buffer&>(kernel6);
```

Figure 5.7: Host Code of Conducting Concurrent Execution of Kernels by Using One Out-of-Order Command Queue

```
for (ncycle=1;ncycle<=ITMAX;ncycle++)
{
        L100_start = lr_tim();
        kernel_l100_cu (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu);
        kernel_l100_h (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_h);
        q.finish();
        L100_end = lr_tim();
        time_spent_l100 = time_spent_l100 + (L100_end - L100_start);

        L100_pc_start = lr_tim();
        kernel_l100_pc (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_cu, buffer_h);
        q.finish();
        L100_pc_end = lr_tim();
        time_spent_l100_pc = time_spent_l100_pc + (L100_pc_end - L100_pc_start);

        L200_start = lr_tim();
        kernel_l200_u (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_cu, buffer_h);
        kernel_l200_p (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_p, buffer_cu, buffer_h);
        q.finish();
        L200_end = lr_tim();
        time_spent_l200 = time_spent_l200 + (L200_end - L200_start);

        L200_pc_start = lr_tim();
        kernel_l200_pc (cl::EnqueueArgs(q, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p);
        q.finish();
        L200_pc_end = lr_tim();
        time_spent_l200_pc = time_spent_l200_pc + (L200_pc_end - L200_pc_start);
        time = time + dt;
}
```

Figure 5.8: Host Code of Executing Kernels Concurrently by Using One Out-of-Order Command Queue

Figure 5.7 and Figure 5.8 shows how this can be implemented using one out-of-order command queue in host code. Kernel L100 is broken down into two separate kernels

called L100_cu and L100_h for better parallelization. Because the computation of array cu and h can be conducted simultaneously. The same method has been applied to kernel L200 as well, where it is broken down into kernel L200_u and L200_p.

```
std::vector<cl::Device> devices = xcl::get_xil_devices();
cl::Device device = devices[0];
std::string device_name = device.getInfo<CL_DEVICE_NAME>();

cl::Context context(device);
cl::CommandQueue q1(context, device);
cl::CommandQueue q2(context, device);

std::string binaryFile = xcl::find_binary_file(device_name,"simple_shallow_kernel");
cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
devices.resize(1);
cl::Program program(context, devices, bins);

cl::Kernel kernel1(program,"l100_cu", &err);
auto kernel_l100_cu = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel1);
cl::Kernel kernel2(program,"l100_h", &err);
auto kernel_l100_h = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel2);
cl::Kernel kernel3(program,"l200_u", &err);
auto kernel_l200_u = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel3);
cl::Kernel kernel4(program,"l200_p", &err);
auto kernel_l200_p = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel4);
cl::Kernel kernel5(program,"l100_pc_cu", &err);
auto kernel_l100_pc_cu = cl::KernelFunctor<cl::Buffer&>(kernel5);
cl::Kernel kernel6(program,"l100_pc_h", &err);
auto kernel_l100_pc_h = cl::KernelFunctor<cl::Buffer&>(kernel6);
cl::Kernel kernel7(program,"l200_pc_u", &err);
auto kernel_l200_pc_u = cl::KernelFunctor<cl::Buffer&>(kernel7);
cl::Kernel kernel8(program,"l200_pc_p", &err);
auto kernel_l200_pc_p = cl::KernelFunctor<cl::Buffer&>(kernel8);
```

Figure 5.9: Host Code of Conducting Concurrent Execution of Kernels by Using Two In-Order Command Queue

```
for (ncycle=1;ncycle<=ITMAX;ncycle++)
{
        L100_start = lr_tim();
        kernel_l100_cu (cl::EnqueueArgs(q1, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu);
        kernel_l100_h (cl::EnqueueArgs(q2, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_h);
        kernel_l100_pc_cu (cl::EnqueueArgs(q1, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_cu);
        kernel_l100_pc_h (cl::EnqueueArgs(q2 cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_h);
        q1.finish();
        q2.finish();
        L100_end = lr_tim();
        time_spent_l100 = time_spent_l100 + (L100_end - L100_start);

        L200_start = lr_tim();
        kernel_l200_u (cl::EnqueueArgs(q1, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_cu, buffer_h);
        kernel_l200_p (cl::EnqueueArgs(q2, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_p, buffer_cu, buffer_h);
        kernel_l200_pc_u (cl::EnqueueArgs(q1, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u);
        kernel_l200_pc_p (cl::EnqueueArgs(q2, cl::NDRange(1,1,1), cl::NDRange(1,1,1)), buffer_p);
        q1.finish()
        q2.finish();
        L200_end = lr_tim();
        time_spent_l200 = time_spent_l200 + (L200_end - L200_start);
        time = time + dt;
}
```

Figure 5.10: Host Code of Executing Kernels Concurrently by Using Two In-Order Command Queue

Figure 5.9 and Figure 5.10 demonstrates how concurrent execution of kernels can be achieved by using two in-order command queues. In this further optimized version, kernel L100_pc is broken down into two separate kernels called L100_pc_cu and L100_pc_h for better parallelization. Because the periodic continuation operation of array cu and h can be conducted simultaneously. The same method has been applied to kernel L200_pc as well, where it is broken down into kernel L200_pc_u and L200_pc_p.

All array-cu-related computations are put into one command queue, while all array- h-related computations are put into another command queue. The situation of the computation of array u and array p is similar.

```
__attribute__ ((vec_type_hint(float)))
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100(__global float * __restrict u, __global float * __restrict p, __global float * __restrict cu)
{
        __local float u_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float p_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float cu_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        async_work_group_copy(cu_buff, cu, wg_size, 0);
        for (i=0;i<M;i++)
        {
                for (j=0;j<N;j++)
                {
                        cu_buff[(i + 1)*M_LEN + (j)] = .33 * (p_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j+1)]) + u_buff[(i + 1)*M_LEN + (j)];
                }
        }
        async_work_group_copy(cu, cu_buff, wg_size, 0);
        return;
}
```

Figure 5.11: Kernel Code of Kernel L100_cu Being Optimized by Using Keyword "__restrict"

Figure 5.11 shows how to use the "__restrict" keyword in kernel L100_cu. The situation in all other kernels is similar. This experiment is based on the experiment conducted in Section 5.3.

5.5. *Iteration 9*: Dataflow and Function Inline

This experiment is investigating the option of pipelining the function calls in each kernel by using directive "xcl_dataflow" and achieving function inline by using directive "always_inline" as mentioned in Section 2.6.1. In order to apply this optimization to the kernel code, some modifications need to be made by packing the async_work_group_copy() function and computation code into function calls. It is worth noting that all the periodic continuation kernels cannot be optimized using "xcl_dataflow" directive.

```
void write_u_p(__global float * __restrict u, __local float *u_buff, __global float * __restrict p, __local float *p_buff)
{
        async_work_group_copy(u, u_buff, wg_size, 0);
        async_work_group_copy(p, p_buff, wg_size, 0);
}
```

Figure 5.12: Kernel Code of Packing async_work_group_copy() into "write_u_p" Function

```
void read_u_p(__global float * __restrict u, __local float *u_buff, __global float * __restrict p, __local float *p_buff)
{
        async_work_group_copy(u_buff, u, wg_size, 0);
        async_work_group_copy(p_buff, p, wg_size, 0);
}
```

Figure 5.13: Kernel Code of Packing async_work_group_copy() into "read_u_p" Function

```
void cal_cu(__local float *cu_buff, __local float *p_buff, __local float *u_buff)
{
        int i,j;
        for (i=0;i<M;i++)
        {
                for (j=0;j<N;j++)
                {
                        cu_buff[(i + 1)*M_LEN + (j)] = .33 * (p_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j+1)]) + u_buff[(i + 1)*M_LEN + (j)];
                }
        }
}
```

Figure 5.14: Kernel Code of Packing Array Update Code into "Calculation" Function

```
__attribute__ ((vec_type_hint(float)))
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__attribute__ ((xcl_dataflow))
__attribute__ ((always_inline))
__kernel void l100_cu(__global float * __restrict u, __global float * __restrict p, __global float * __restrict cu)
{
        __local float u_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float p_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float cu_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        read_u_p(u, u_buff, p, p_buff);
        cal_cu(cu_buff, p_buff, u_buff);
        write_cu(cu, cu_buff);
        return;
}
```

Figure 5.15: Kernel Code of Using Function Calls in Kernel L100_cu, as well as Function Calls Pipelining and Function Inline

Figure 5.12 demonstrates how the async_work_group_copy() functions are packed into "write_u_p" function. The situations in other "write" functions are similar. Figure 5.13 demonstrates how the async_work_group_copy() functions are packed into "read_u_p" function. The situations in other "read" functions are similar. Figure 5.14 shows how the array update codes are packed into "calculation" functions. Figure 5.15 explains how to replace the original code with function calls in kernel L100_cu, as well as the usage of directive "xcl_dataflow" and "always_inline". Situation in all other array update kernels are similar. This experiment is based on the experiment conducted in Section 5.4.

5.6. *Iteration 10*: Merging Array Update Kernel with Periodic Continuation Kernel

This experiment is investigating the option of merging the array update kernel with the periodic continuation kernel, as mentioned in Section 2.6.1. In the previous design, array update operation and periodic continuation operation are conducted in two different kernels, which means data will be transferred between global memory and local memory four times, as demonstrated in Figure 4.8, Figure 4.9 and Figure 5.10. This is unnecessary because periodic continuation can be conducted right after array update is complete, since all the whole array is cached in BRAM. By merging array update kernel with periodic continuation kernel, data only needs to be transfer twice between global memory and local memory.

```
for (ncycle=1;ncycle<=ITMAX;ncycle++)
{
        L100_start = lr_tim();
        kernel_l100_cu (cl::EnqueueArgs(q1, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_cu);
        kernel_l100_h (cl::EnqueueArgs(q2, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_p, buffer_h);
        q1.finish();
        q2.finish();
        L100_end = lr_tim();
        time_spent_l100 = time_spent_l100 + (L100_end - L100_start);

        L200_start = lr_tim();
        kernel_l200_u (cl::EnqueueArgs(q1, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_u, buffer_cu, buffer_h);
        kernel_l200_p (cl::EnqueueArgs(q2, cl::NDRange(1,1,1), cl::NDRange(1,1,1)),buffer_p, buffer_cu, buffer_h);
        q1.finish();
        q2.finish();
        L200_end = lr_tim();
        time_spent_l200 = time_spent_l200 + (L200_end - L200_start);
}
```

Figure 5.16: Host code that shows Periodic Continuation Kernels are merged with Array Update Kernels

```
__attribute__ ((vec_type_hint(float)))
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__kernel void l100_cu( __global float * __restrict u,
                       __global float * __restrict p,
                       __global float * __restrict cu)
{
        __local float u_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float p_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        __local float cu_buff[wg_size] __attribute__((xcl_array_partition(cyclic,2,1)));
        async_work_group_copy(u_buff, u, wg_size, 0);
        async_work_group_copy(p_buff, p, wg_size, 0);
        int i,j;
        for (i=0;i<M;i++)
        {
                for (j=0;j<N;j++)
                {
                        cu_buff[(i + 1)*M_LEN + (j)] = .33 * (p_buff[(i + 1)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j)] + p_buff[(i)*M_LEN + (j+1)]) + u_buff[(i + 1)*M_LEN + (j)];
                }
        }
        int i1,j1;
        __attribute__((opencl_unroll_hint(2)))
        for (j1=0;j1<N;j1++)
        {
                cu_buff[(0)*M_LEN + (j1)] = cu_buff[(M)*M_LEN + (j1)];
        }
        __attribute__((opencl_unroll_hint(2)))
        for (i1=0;i1<M;i1++)
        {
                cu_buff[(i1 + 1)*M_LEN + (N)] = cu_buff[(i1 + 1)*M_LEN + (0)];
        }
        cu_buff[(0)*M_LEN + (N)] = cu_buff[(M)*M_LEN + (0)];
        async_work_group_copy(cu, cu_buff, wg_size, 0);
        return;
}
```

Figure 5.17: Kernel Code of Periodic Continuation Operation merged into Kernel L100_cu

Figure 5.16 demonstrates the main computation loop in the host code of ***iteration 10***, where periodic continuation kernels no longer exist. While Figure 5.17 shows how this is implemented in kernel L100_cu. Situations in other kernels are similar. This experiment is based on the experiment conducted in Section 5.4, because periodic continuation operation cannot be optimized using function calls pipelining and function inline.

5.7. Summary

The experiments described in this chapter are summarized in the following table,

Optimization	Array Partitioning	Data Vectorization	Overlapping Data Transfer and Computation	Restrict Keyword and Kernel Concurrent Execution	Dataflow and Function inline	Kernel Merging
Iteration 5	√	X	X	X	X	X
Iteration 6	√	√	X	X	X	X
Iteration 7	√	√	√	X	X	X
Iteration 8	√	√	√	√	X	X
Iteration 9	√	√	√	√	√	X
Iteration 10	√	√	√	√	X	√

Table 5.1: Summary of the Experiments Described in Chapter 5

Chapter 6 Experimental Data Analysis

This project are evaluated using an ARM CPU-FPGA heterogeneous system on a single Zynq® UltraScale+™ MPSoC ZCU102 board. Evaluation conducted with multiple Zynq® UltraScale+™ MPSoC ZCU102 boards is left for future work.

Both speedup, hardware resource utilization and latency information are considered during the evaluation. The speedup can be obtained by using the following formula,

$$\text{Speedup} = T_{Baseline} / T_{Optimized}$$

where $T_{Baseline}$ is the execution time of the baseline code, while $T_{Optimized}$ is the execution time of the optimized code.

The information of hardware resources utilization as well as latency can be obtained from the system estimate report and HLS report. Interpretation of the information is discussed in Section 6.1. Power consumption of the application is discussed in Section 6.8.

The results are summarised in diagrams and tables indicating speedup, latency and hardware resource utilization.

In terms of performance, the achieved performance of the simplified shallow water weather & climate forecasting application implemented in this project is compared against the performance of an existing, sequential C version of simplified shallow water application executing on modern CPU. The performance of the simplified shallow water application runs on FPGA is also compared with the performance of the application that runs on modern GPGPU. (Pappas, 2012) also provides some performance data of the original shallow water application executing on CPUs and GPGPUs.

What's more, since several increasingly sophisticated, simplified shallow water applications are implemented, the speedup, latency and hardware resource utilization are compared incrementally against each implementation.

All the experiments are executed three times, the execution time is the average of the three runs. The standard deviation is also calculated. If the results are not consistent, meaning that the standard deviation is not small enough, the experiment will be run two more times. The result of the first run is always be forfeited to avoid any start-up overhead. The execution time of the main computation loop as well as the whole application are logged, the execution time of each kernel, are logged as well if it is applicable and necessary. The hardware resource utilization, latency information as well as the loop information, which can be obtained from the system estimate report and HLS report, are also recorded.

6.1. Latency and Loop Information Interpretation

This section explains how to interpret the latency information and the loop information provided in the system estimate report and HLS report. The latency and loop information of kernel L100 of the baseline code serves as an example here.

Kernel	Start Interval	Best case	Average case	Worst case
L100	17190	17189	17189	17189

Table 6.1: Latency Information of Kernel L100 of the Baseline Code

Kernel	Loop	Min latency	Max latency	Iteration latency	Achieved II	Target II	Trip count	Pipelined
L100	calc**	17187	17187	292	4	1	4225	yes

**: calc stands for the update of array cu and array h.

Table 6.2: Loop Information of Kernel L100 of the Baseline Code

Table 6.1 and Table 6.2 demonstrate the latency and loop information of kernel L100 of the baseline code. The terms used in Table 6.1 are defined in Section 2.4. In Table 6.1, the start interval is very close to the best, average and worst case of latency, which means the functions in kernel L100 are not overlapped by default. Function calls overlapping is discussed in the definition of dataflow directive in Section 2.6.1. The minimum and maximum latency in Table 6.2 is very close to the best, average and worst case latency in Table 6.1, although they are not the same. Here pipelined indicates the inner loop is pipelined

It should be noted that, in a pipelined loop,

$$\text{Latency}_{\text{total}} \approx [\text{ Achieved II} * (\text{Trip count} - 1)] + \text{Iteration latency}$$

This is because the achieved II represent the number of clock cycles each iteration (except the first one) needed to produce the result in a pipelined loop. However, for the result of the first iteration it will always be the number of iteration latency. This equation can be verified by using the data in Table 6.2. The right side of the equation equals,

$$[\ 4 * (4225 - 1)\ ] + 292 = 17188$$

which is very close to total latency 17187.

6.2. The Scalability Model

A simple scalability model is presented here which aims to discover the relationship between the hardware resource utilization and speedup obtained for each implementation, versus the baseline. This model is useful when it comes to the situation that one optimization method needs to be selected from multiple optimization methods by a developer.

The basic idea behind this model is an IP block is generated based on the code which consumes a certain amount of hardware resource. The total amount of hardware resource on a single FPGA is limited, so the number of IP blocks which can be generated and included can be calculated. Assuming all the generated IP blocks can run in parallel, after the execution time of each IP block on a given problem size is obtained, the throughput, which is defined below, can then be acquired. Take the baseline code as an example:

Kernel	FF	LUT	DSP	BRAM_18K
L100	3692	4003	16	6
L200	3682	5905	16	2
L100_pc	3440	4194	0	2
L200_pc	3440	4194	0	2
Total	14254	18296	32	12

Table 6.3: Hardware Resource Utilization of the Baseline Code

Table 6.3 demonstrates the amount of hardware resource needed for each kernel. From Table 6.3 it can be deduced that a copy of IP block that consists of a total number of 14254 FFs, 18296 LUTs, 32 DSPs and 12 BRAM_18Ks is generated. The FPGA used in this project has a total number of 548160 FFs, 274080 LUTs, 2520 DSPs and 1824 BRAM_18Ks. Hence, at most 14 copies of such IP block can be generated. Since the

execution time of the baseline code is around 106 seconds with a problem size of 65 * 65 elements, which implies a throughput of approximately 40 elements per second. With 14 copies of IP blocks and perfect scalability, a throughput of (65 * 65 * 14) / 106 which is around 558 elements per second can be achieved.

In conclusion, the scalability model can be summarized as follow,

$$N_{hardware} = \min\left(\left\lfloor\frac{FF_{total}}{FF_{single}}\right\rfloor, \left\lfloor\frac{LUT_{total}}{LUT_{single}}\right\rfloor, \left\lfloor\frac{DSP_{total}}{DSP_{single}}\right\rfloor, \left\lfloor\frac{BRAM_{total}}{BRAM_{single}}\right\rfloor\right)$$

$$\text{Throughput} = (\text{ Problem size} * N_{hardware}) / T$$

where $N_{hardware}$ represent the number of IP block being generated. FF_{total}, LUT_{total}, DSP_{total} and $BRAM_{total}$ is the total number of FF, LUT, DSP and BRAM on a single FPGA respectively. FF_{single}, LUT_{single}, DSP_{single} and $BRAM_{single}$ is the number of FF, LUT, DSP and BRAM a generated IP block consists respectively. T represents the execution time of main computation loop within one generated IP block.

6.3. Why Emulator is unreliable

This section explains in detail, with the help of experimental data, why the emulator is unreliable in terms of execution time measurement.

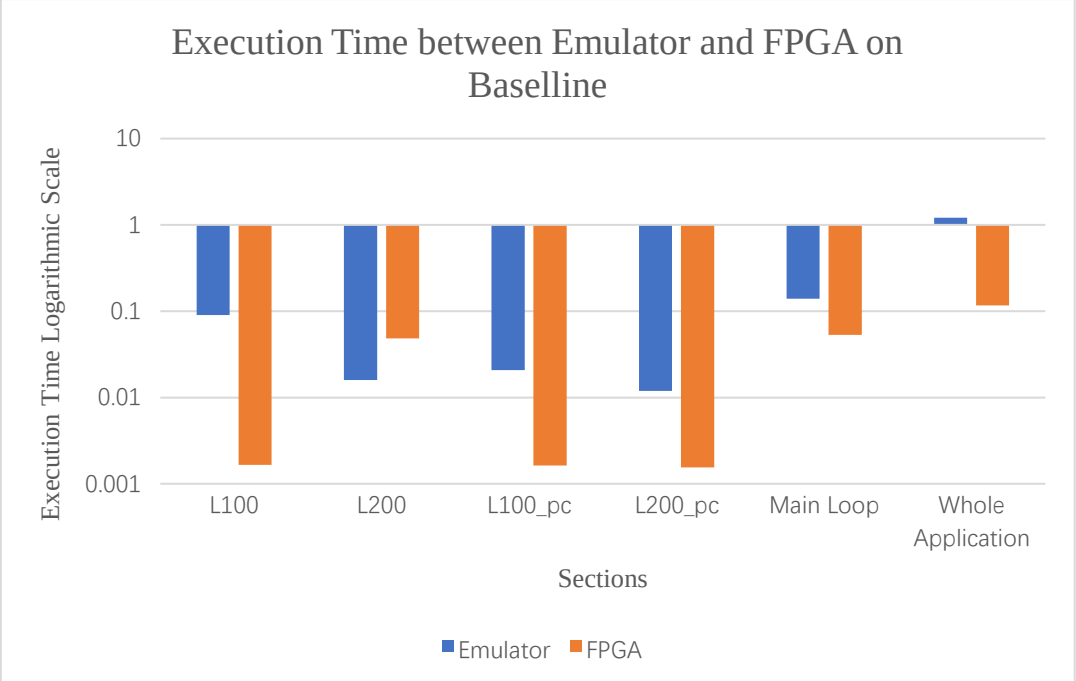


Diagram 6.1: Execution Time of Different Section of the Baseline Code between Emulator and FPGA

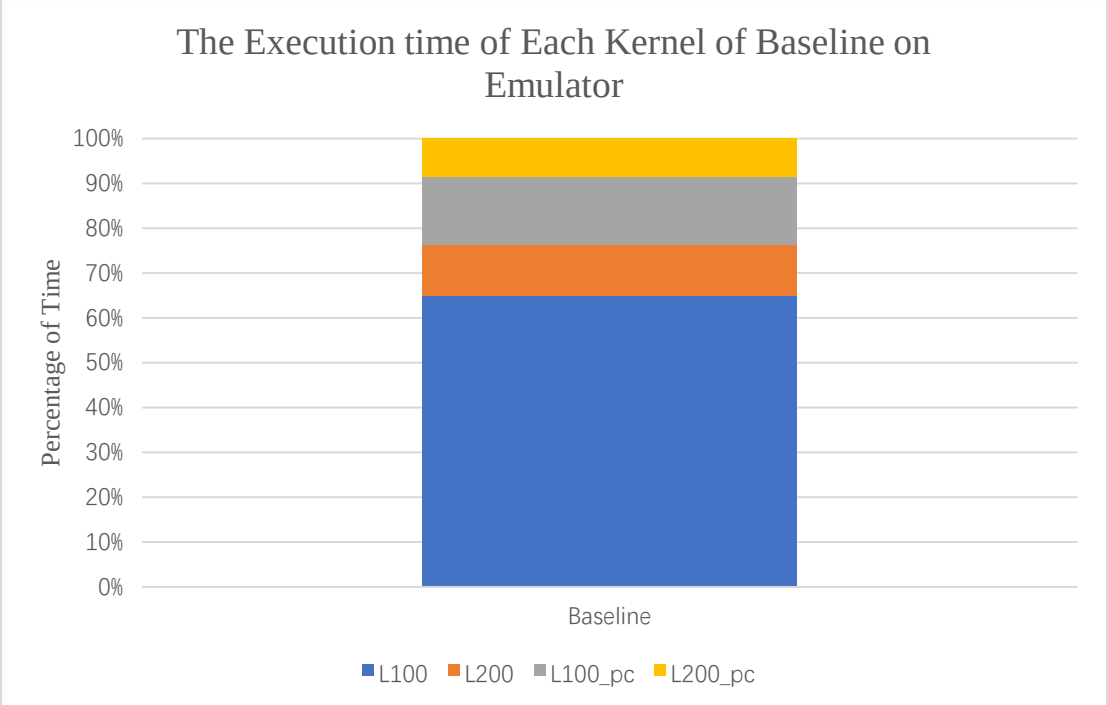


Diagram 6.2: The Execution Time of each Kernel of the Baseline Code on Emulator

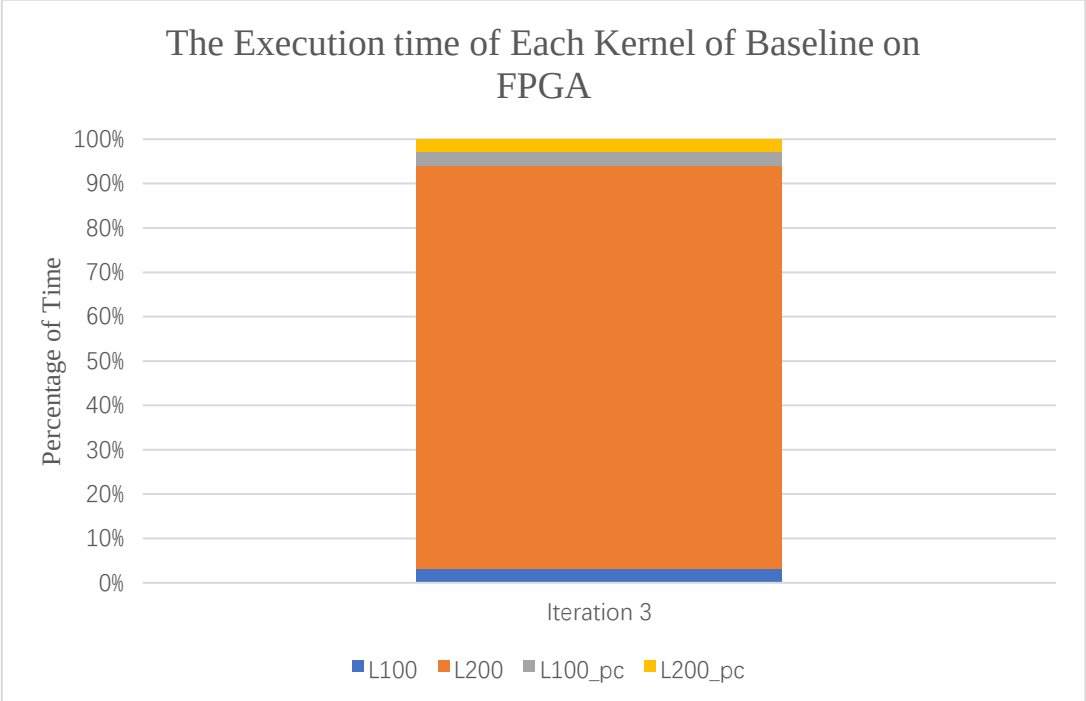


Diagram 6.3: The Execution Time of each Kernel of the Baseline Code on FPGA

Diagram 6.1 demonstrates the execution time of each kernel, as well as the main loop and the whole application, when executing the baseline code on both emulator and FPGA. It is obvious that the difference between the execution time measured on FPGA and the one measured on emulator is significant, in every kernel as well the time of the main loop and the whole application. Hence, the execution time measured on emulator is not accurate.

Diagram 6.2 and Diagram 6.3 provide some more evidence regarding why the execution time measured on emulator is not accurate. Diagram 6.2 describes the execution time of each kernel when executing the baseline code on emulator. Clearly kernel L100 is the one that dominates the execution time. Diagram 6.3 demonstrates the execution time of each kernel when executing the baseline code on FPGA. In this case, it is obvious that kernel L200 is the one that dominates the execution time. When cross referencing with the baseline system estimate report, it can be found out that the average latency for kernel L200 is significantly larger than the other kernels, which means Diagram 6.3 should be the accurate one instead of Diagram 6.2.

Diagram 6.4: Execution Time of Different Section between the Baseline Code and *Iteration 3* on Emulator

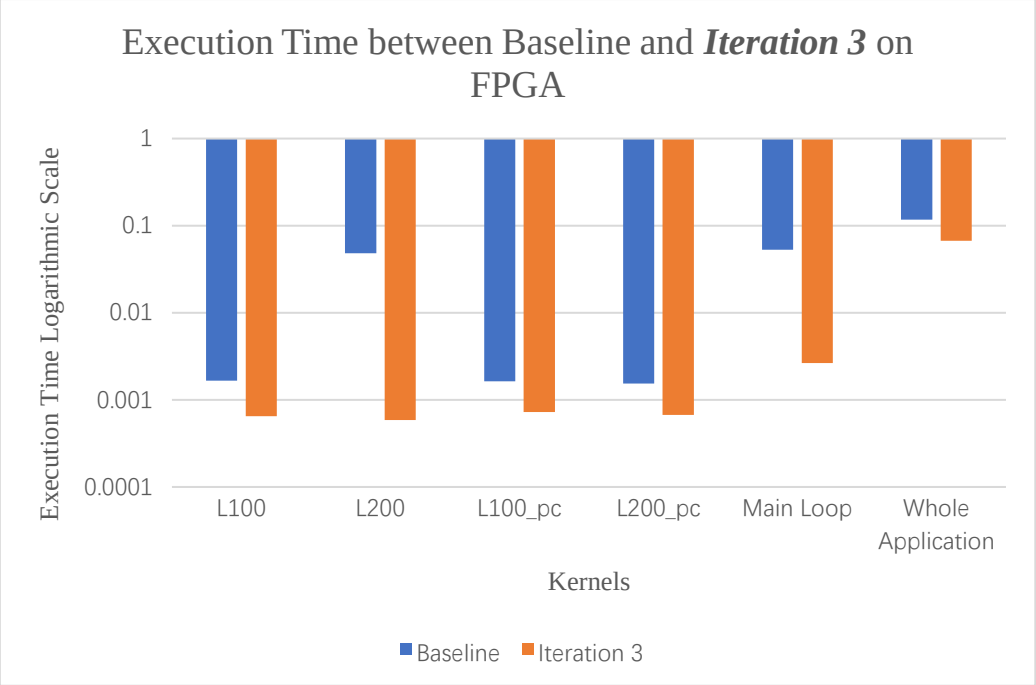
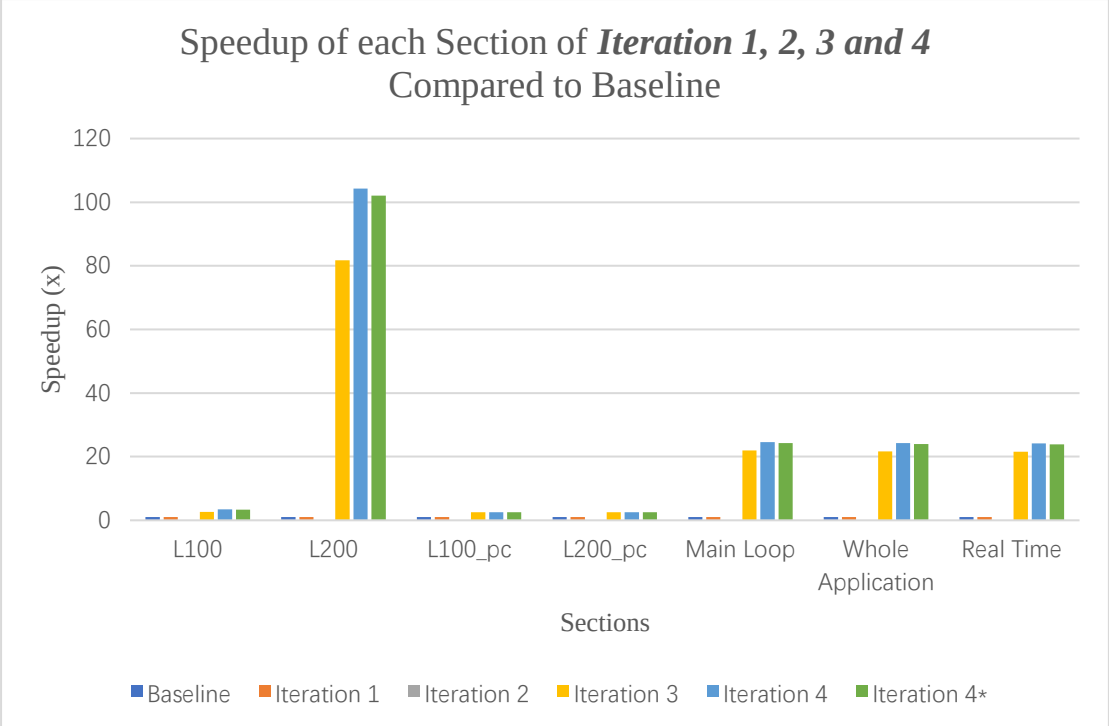


Diagram 6.5: Execution Time of Different Section between the Baseline Code and *Iteration 3* on FPGA

Diagram 6.4 demonstrates the execution time between baseline and ***iteration 3*** on emulator, of each kernel as well the main loop and the whole application. Diagram 6.5 presents the execution time between baseline and ***iteration 3*** on FPGA, of each kernel as well the main loop and the whole application. ***Iteration 3*** is an optimization that significantly improves the performance, this can be seen from Diagram 6.5, the execution time of each kernel as well as the main loop and the whole application is shortened. However, this is not the case in emulator. As Diagram 6.4 shows, there is no significant change in execution time after applying ***iteration 3***. Even there is a change in execution time, it only makes the performance becomes worse. Therefore, the emulator should not be used as an indicator of performance improvement. Because performance improvement on FPGA doesn't necessarily means performance improvement on emulator.

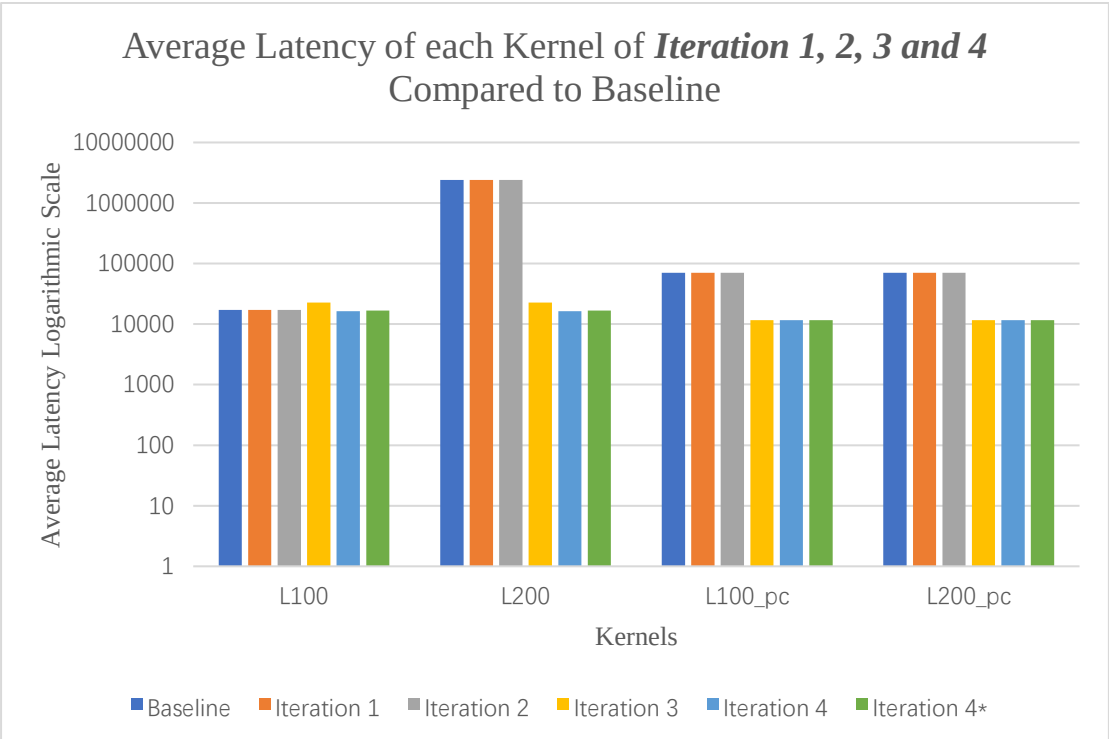
6.4. Data from Experiments with Basic Optimizations

This section analyses the speedup, hardware resource utilization, latency and loop information from the experiments conducted in Chapter 4.



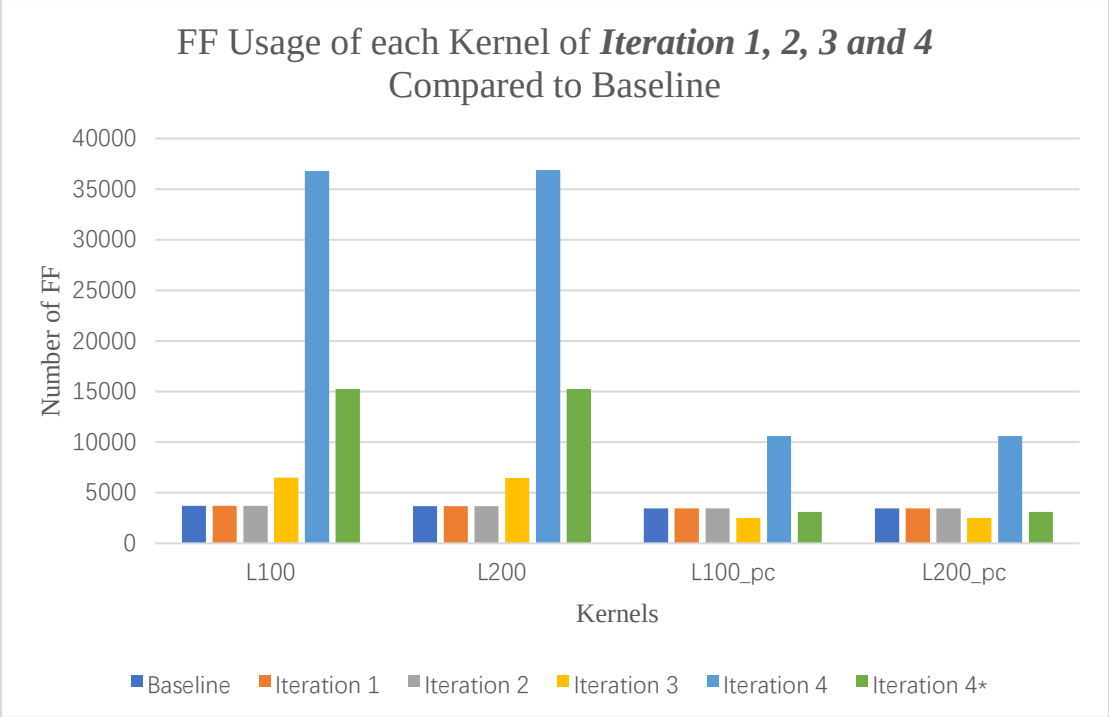
Iteration 4* means unrolling the loops in each kernel using a factor of 8.

Diagram 6.6: Speedup of each Section of *Iteration 1, 2, 3 and 4* Compared to Baseline



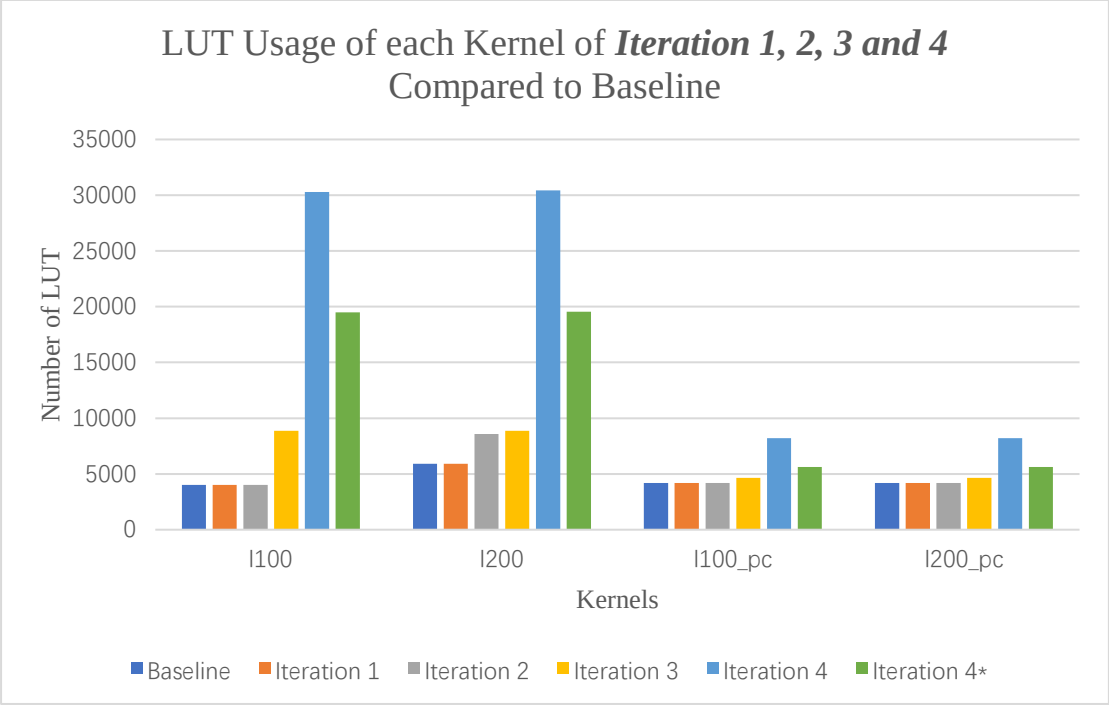
Iteration 4* means unrolling the loops in each kernel using a factor of 8.

Diagram 6.7: Average Latency of each Kernel of *Iteration 1, 2, 3 and 4* Compared to Baseline



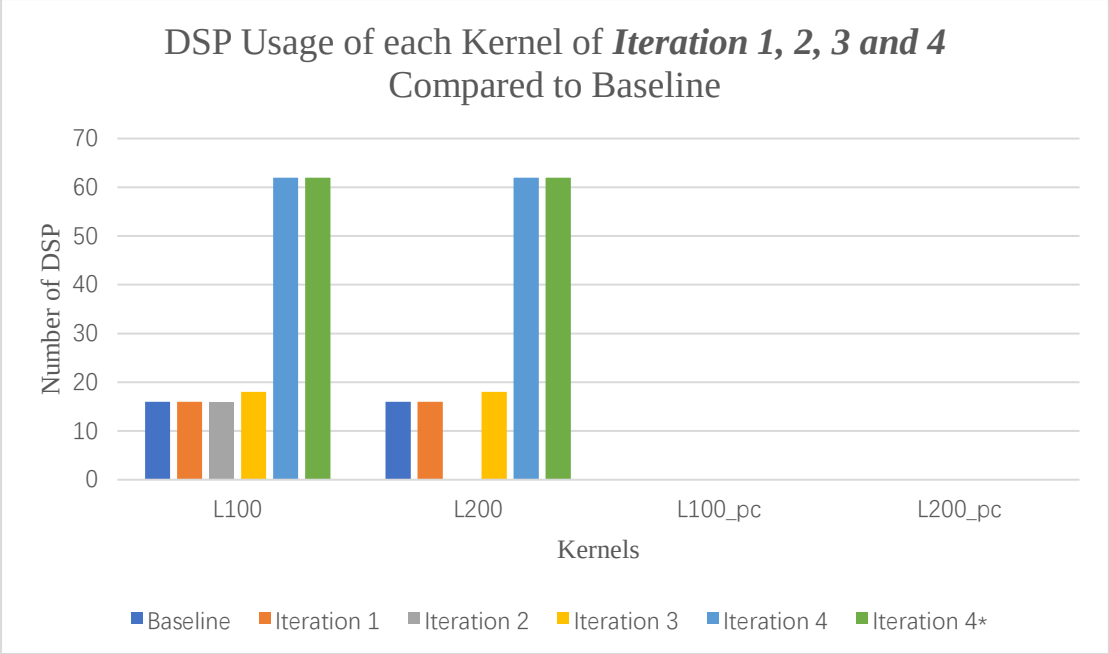
Iteration 4* means unrolling the loops in each kernel using a factor of 8.

Diagram 6.8: FF Usage of each Kernel of *Iteration 1, 2, 3 and 4* Compared to Baseline



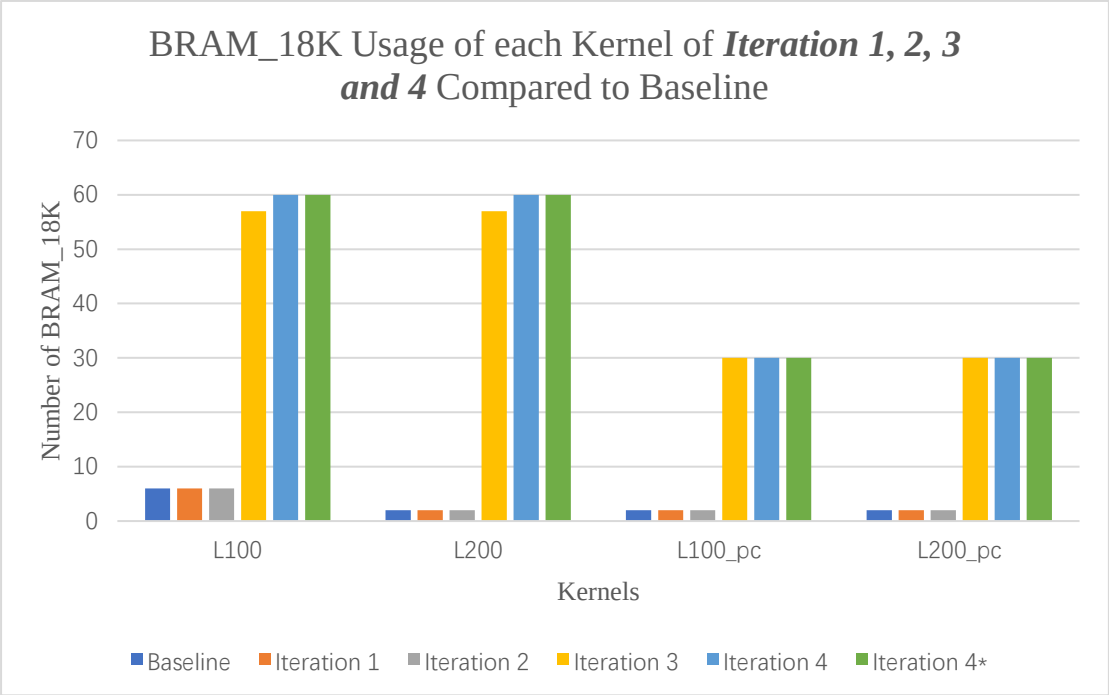
Iteration 4* means unrolling the loops in each kernel using a factor of 8.

Diagram 6.9: LUT Usage of each Kernel of *Iteration 1, 2, 3 and 4* Compared to Baseline



Iteration 4* means unrolling the loops in each kernel using a factor of 8.

Diagram 6.10: DSP Usage of each Kernel of *Iteration 1, 2, 3 and 4* Compared to Baseline



Iteration 4* means unrolling the loops in each kernel using a factor of 8.

Diagram 6.11: BRAM_18K Usage of each Kernel of *Iteration 1, 2, 3 and 4* Compared to Baseline

Kernel	Loop	Min latency	Max latency	Iteration latency	Achieved II	Target II	Trip count	Pipelined
L200	calc*	2395575	2395575	567	N/A	N/A	4225	No
L100_pc	pc1*	34905	34905	537	N/A	N/A	N/A	No
	pc2**	34905	34905	537	N/A	N/A	N/A	No
L200_pc	pc1	34905	34905	537	N/A	N/A	N/A	No
	pc2	34905	34905	537	N/A	N/A	N/A	No

calc* means the loop for updating the array.

pc1* means the first periodic continuation loop.

pc2** means the second periodic continuation loop.

Table 6.4: Partial Loop Information of *Iteration 2*

Kernel	Loop	Min latency	Max latency	Iteration latency	Achieved II	Target II	Trip count	Pipelined
L100	calc*	8471	8471	24	2	1	4225	Yes
L200	calc	8471	8471	24	2	1	4225	Yes
L100_pc	pc1*	65	65	2	1	1	65	Yes
	pc2**	65	65	2	1	1	65	Yes
L200_pc	pc1	65	65	2	1	1	65	Yes
	pc2	65	65	2	1	1	65	Yes

calc* means the loop for updating the array.

pc1* means the first periodic continuation loop.

pc2** means the second periodic continuation loop.

Table 6.5: Partial Loop Information of *Iteration 3*

Diagram 6.6 shows the speedup of each kernel, as well as the main loop, whole application and real time of ***iteration 1, 2, 3 and 4*** against the baseline. Diagram 6.7 shows the average latency of each kernel of baseline, ***iteration 1, 2, 3 and 4***. Diagram 6.8 (FF), Diagram 6.9 (LUT), Diagram 6.10 (DSP) and Diagram 6.11 (BRAM_18K) show the hardware resource utilization of each kernel when executing baseline code as well as ***iteration 1, 2, 3 and 4***.

By analysing the baseline code, which is mentioned in Section 2.5.2 and Section 4.1, the non-parallel code overhead as well the memory access overhead is the most significant overhead initially, because none of the code is parallelized and all the data are stored in the slow off-chip memory in the baseline. Synchronization overheads also

exist because kernels need to be synchronized for measuring the execution time. However, these synchronizations are inevitable and they shouldn't be the main overhead. There shouldn't be any load imbalance overhead and scheduling overhead either.

In order to tackle the non-parallel code and memory access overheads, the compiler option of -O3 optimization is first applied in ***iteration 1***. Surprisingly, the -O3 optimization doesn't bring any significant change in terms of speedup, in every kernel as well as the main loop, whole application and real time. According to Diagram 6.7, there is no significant change in the average latency of each kernel either. The reason behind this is yet unknown and will be left for future work. Hence, it is decided that the -O3 compiler optimization option is not be used in future experiments, because it needs to be made sure that the optimizations are not introduced by the compiler.

As Diagram 6.7 shows, for baseline and ***iteration 1***, the average latency in kernel L200 as well as kernel L100_pc and L200_pc is significantly higher than kernel L100. Therefore, ***iteration 2*** aims to minimize the non-parallel code overhead in kernel L200, L100_pc and L200_pc by manually pipelining the loops in these kernels. However, there is still no significant change in terms of average latency in kernel L200, L100_pc and L200_pc, and speedup is not obtained for ***iteration 2*** as would be expected, according to the efficient FPGA programming principle mentioned in Section 3.5. The reason why the average latency in kernel L200, L100_pc and L200_pc is not changed significantly is that the loops in these kernels are still not pipelined in the end, despite the "xcl_pipeline_loop" directive being placed manually and explicitly ahead or within the appropriate loop in the kernel code. The evidence can be found in Table 6.4. It is believed that one of the reasons why the compiler is unable to pipeline the loops is because of the memory access latency overhead. The investigation of the other reasons will be left for future work. It should be noted that the experiment to completely unroll the loops in kernel L200, L100_pc and L200_pc is not conducted due to the extremely long and impractical compilation time required.

Iteration 3 aims to tackle the memory access overhead by caching all the data into the on-chip BRAM. In order to minimize the data transfer overhead, burst memory transfer

is used as well, which enables the read or write of four words in each clock cycle. These optimization methods are mentioned in Section 2.6.1. In ***iteration 3***, the speedup of kernel L200 is improved by around 81x compared to baseline, while the speedup of main loop, whole application and real time is improved by around 21x compared to baseline, according to Diagram 6.6. The speedup of kernel L100, L100_pc and L200_pc is improved by around 2.5x as well. The average latency of kernel L200 is improved by around 106x, while the average latency of kernel L100_pc and L200_pc is improved by around 6x. However, although the speedup of kernel L100 is improved, its average latency is worsened by around 32%. Also, it should be noted that after caching data in on-chip BRAM, all the loops in kernel L100_pc and L200_pc, as well as the inner loop in kernel L100 and L200 are pipelined automatically by the compiler. This can be seen in Table 6.5. The reason why they can be pipelined by the compiler might be the significant minimization of the memory access overhead. Therefore, the local memory as well as the burst memory transfer will continue to be used in the future experiments. There is no need to manually pipeline the loops as well.

Iteration 4 is intended to parallelize the loops by unrolling them. This optimization method is mentioned in Section 2.6.1. In this iteration, the loops are unrolled in two ways, one using an unrolling factor of 8 (***iteration 4****), and another that completely unrolls the loops (***iteration 4***). The experiments conducted here are trying to answer the following question, which is better, loop pipelining or loop unrolling? According to Diagram 6.6 and Diagram 6.7, the speedup of kernel L200 is improved by around 104x compared to baseline, while the speedup of main loop, whole application and real time is improved by around 24x compared to baseline, after unrolling the inner loop completely. The speedup of kernel L100 is improved by around 3.4x, while the speedup of kernel L100_pc, L200_pc is improved by around 2.5x. The average latency of kernel L100 and L200 is improved by around 27.6% as well, after unrolling their inner loop completely. It seems that loop unrolling is better than loop pipelining based on the speedup and latency data.

However, when the hardware resource utilization is considered, things are quite different. According to Diagram 6.8, Diagram 6.9 and Diagram 6.10, complete loop unrolling consumes a lot more FF, LUT and DSP compared to loop pipelining. Since

the hardware resources on each FPGA is limited, this is when the scalability problem arises. By utilizing the model introduced in Section 6.2, the data in Table 6.4 can be obtained.

Optimization Iteration	Speedup of Main Loop (x)	Throughput (Number of Elements per Second)	Number of IP Blocks
Iteration 3	21.96232006	8579	10
Iteration 4	24.60873861	2944	3
Iteration 4*	24.31443444	4849	5

Iteration 4* means unrolling the loops in each kernel using a factor of 8.

Table 6.6: Speedup of Main Loop and the Throughput of *Iteration 3 and 4*

According to Table 6.4, even though ***iteration 4*** has the best speedup, its throughput is the worst, due to the significant amount of hardware resource it consumes. In fact, ***iteration 4**** which unrolls the loops using an unroll factor of 8 is a more scalable option. With its speedup very close to ***iteration 4***, ***iteration 4**** has a better throughput because it consumes fewer hardware resources. In conclusion, loop pipelining will be used in the future experiments since it is the most scalable way of parallelizing loops.

Diagram 6.8, Diagram 6.9, Diagram 6.10 and Diagram 6.11 reveal some other interesting facts. According to Diagram 6.8, there is an increment in the use of FF when the loops in kernel L100 and L200 are pipelined (***iteration 3***). However, it is worth noting that when the loops in kernel L100_pc and L200_pc are pipelined (***iteration 3***), the number of FF that is used decreases. What's more, there is significant increase when the loops in each kernel are unrolled (***iteration 4 and 4****). But there is also a decrease in the amount of FF that is used in kernel L100_pc and L200_pc when their loops are unrolled using a factor of 8 (***iteration 4****), compared to baseline. Diagram 6.9 demonstrates the fact that the use of LUT is incremented after the loops in each kernel are pipelined. There is another significant increment (especially in kernel L100 and L200) when the loops are completely unrolled or unrolled using a factor of 8. It can be seen from Diagram 6.10, the use of DSP is tripled after the loops are unrolled in each kernel. It should be noted that kernel L100_pc and L200_pc doesn't use any DSPs because no floating-point computation is done in the periodic continuation kernels. Another interesting fact is that no DSP is used after manually pipelining the loops in

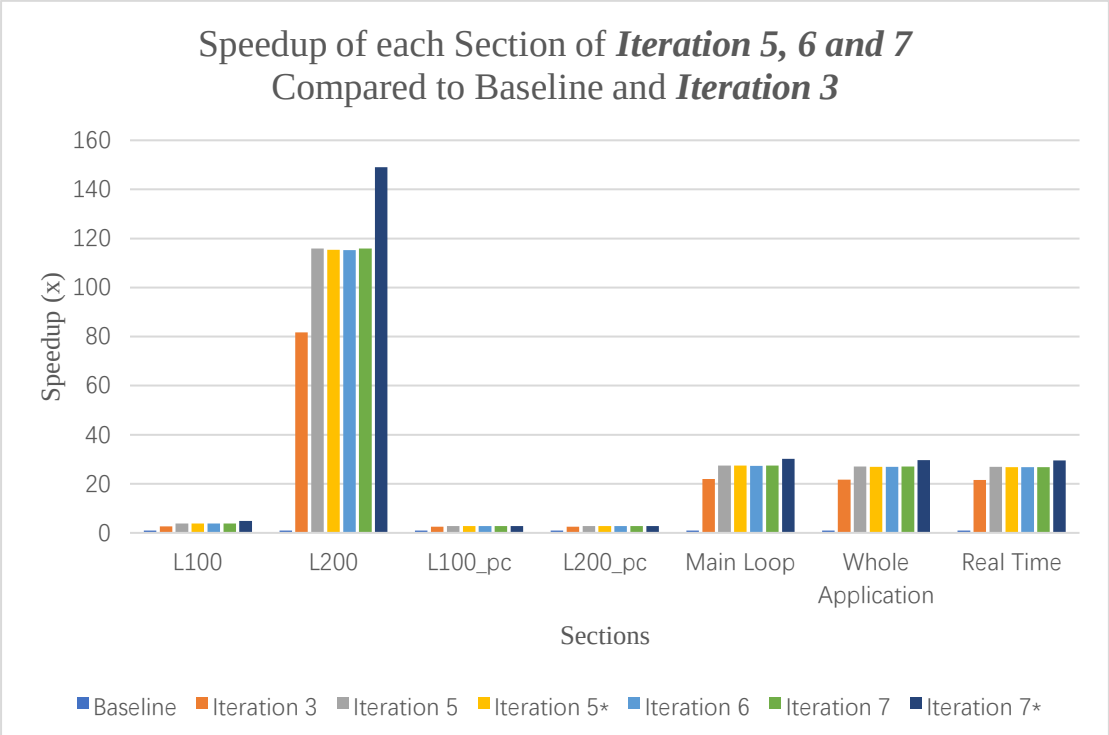
kernel L200. This is probably because the floating-point operations are conducted using LUTs, which is revealed in Diagram 6.9. An in-depth investigation of this will be left for future work. According to Diagram 6.11, there is a significant increment in the amount of BRAM_18K that is used in ***iteration 3 (pipelining), 4 and 4* (unrolling)***. This is because data are cached in the BRAM in these iterations. The next section considers the advanced optimization described in Chapter 5.

6.5. Data from Experiments with Advanced Optimizations

This section analyses the speedup, hardware resource utilization, latency and loop information from the experiments described in Chapter 5.

6.5.1. Iteration 5, 6 and 7

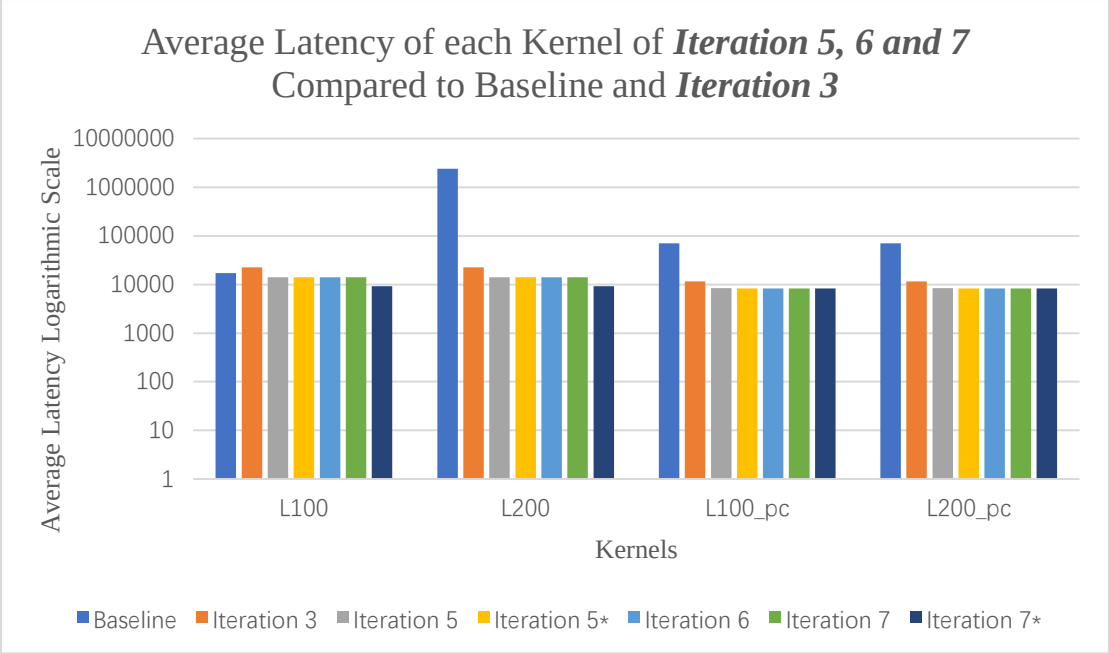
Diagram 6.12 and Diagram 6.13 demonstrates the speedup and average latency of each kernel, as well as the speedup of the main loop, whole application and real time of baseline, ***iteration 3, 5, 6 and 7***. ***Iteration 3*** is included for comparison



Iteration 5* means unrolling the loops in kernel L100_pc and L200_pc using a factor of 2.

Iteration 7* means using fewer async_work_group_copy()

Diagram 6.12: Speedup of each Section of *Iteration 5, 6 and 7* Compared to Baseline and *Iteration 3*



Iteration 5* means unrolling the loops in kernel L100_pc and L200_pc using a factor of 2.

Iteration 7* means using fewer async_work_group_copy()

Diagram 6.13: Average Latency of each Kernel of *Iteration 5, 6 and 7* Compared to Baseline and *Iteration 3*

Kernel	Loop	Min latency	Max latency	Iteration latency	Achieved II	Target II	Trip count	Pipelined
L100_pc	pc1*	32	32	2	1	1	1	Yes
	pc2**	33	33	2	1	1	1	Yes
L200_pc	pc1	32	32	2	1	1	1	Yes
	pc2	33	33	2	1	1	1	Yes

pc1* means the first periodic continuation loop.

pc2** means the second periodic continuation loop.

Table 6.7: Partial Loop Information of *Iteration 5**

Iteration 5 aims to deal with the warning message of "limited memory port" which arises during compilation, as mentioned in Section 5.1, by partitioning the array into multiple physical memories to provide more memory ports, which is mentioned in Section 2.6.1. According Diagram 6.12, the speedup of L200 is improved by around 116x compared to baseline, in ***iteration 5***. The speedup of main loop, whole application and real time is improved by around 27x as well. A decrement in average latency of around 37% can also be observed in kernel L100 and L200 from Diagram 6.13, compared with ***iteration 3***. While the decrement in average latency of the periodic continuation kernels is around 27%. Therefore, all the array will continue to be

partitioned in the future experiments.

Iteration 5* unrolls the loops in kernel L100_pc and L200_pc with a factor 2, to minimize the initial interval, which is mentioned in Section 2.4, to 1. The initial interval is indeed minimized according to Table 6.7. However, no significant improvement is observed in terms of speedup in Diagram 6.12 for any kernel or the main loop, whole application and real time. There is also no significant improvement in average latency in any kernel according to Diagram 6.13.

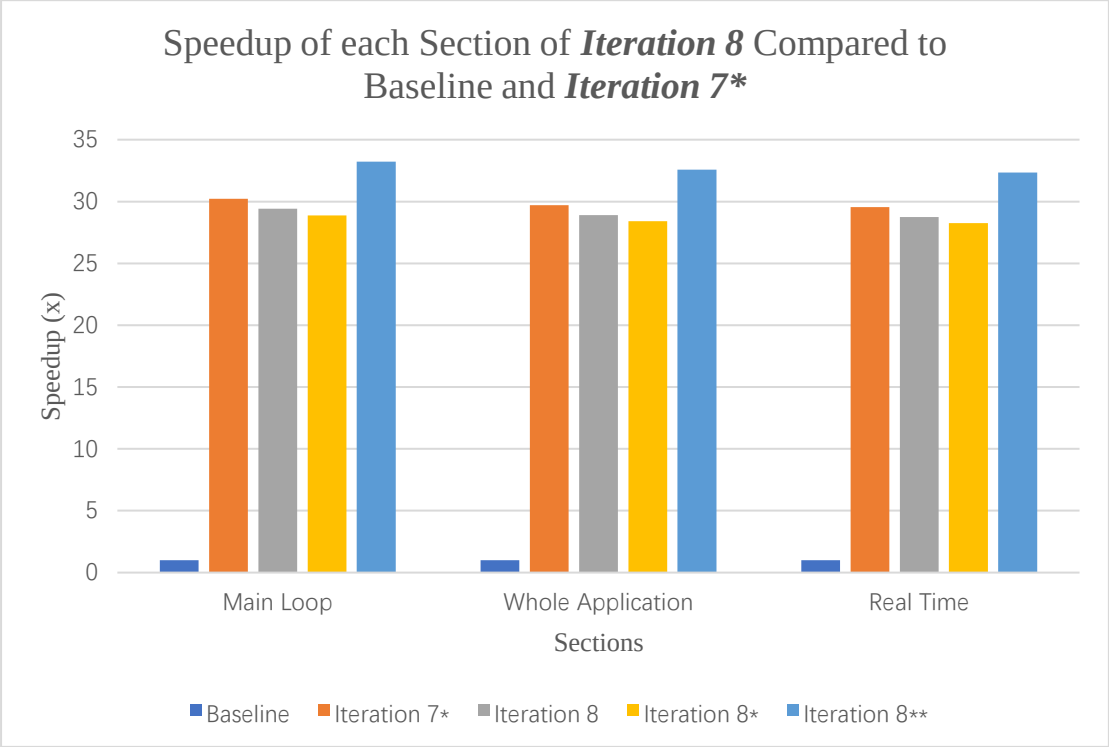
Iteration 6 investigates the option of utilizing the memory bandwidth in a better way by data vectorization, which is mentioned in Section 2.6.1. For simplicity, data vectorization is achieved here by using directive "vec_type_hint" to vectorize the data automatically. However, no significant improvement is observed in terms of speedup in any kernel and main loop, whole application and real time. There is no significant improvement in average latency in any kernel either. Data in Diagram 6.12 and Diagram 6.13 suggest that the automatic data vectorization doesn't appear to work in the simplified shallow water application. The reason behind this might be the complexity of the code which hinders the data from being vectorized automatically. Despite the failure in observing any significant improvement of speedup or average latency, the automatic data vectorization will still be kept and used in the future experiments.

Iteration 7 applies the option of overlapping the data transfer between host and device with kernel computation, which is mentioned in Section 2.6.2, to improve performance. Again, no significant improvement is observed in terms of speedup in any kernel or main loop, whole application and real time. The reason behind this might be, that the speedup brought by the overlapping is counteracted by the newly-introduced synchronization overhead, as well as the unaligned memory allocator, which might lead to more memory copy. The enqueueMigrateMemObjects() function which achieves the overlapping needs to be synchronized each time it finishes execution in order to obtain the correct result, which is mentioned in Section 5.3. However, function enqueueWriteBuffer() and enqueueReadBuffer(), the ones that are replaced by enqueueMigrateMemObjects() don't need to be synchronized. It should be noted that the hardware resource utilization data cannot reflects whether the optimization of

overlapping of data transfer between host and device with kernel computation is working or not. This is because this optimization is a host optimization which means it won't change the generated system estimate report. Despite the failure in observing any significant improvement of speedup, the overlapping of data transfer and kernel computation will still be kept and used in the future experiments.

Iteration 7* optimizes the code by using fewer async_work_group_copy() calls as mentioned in Section 2.6.1. It should be noted that async_work_group_copy() function is treated as a loop by the compiler. Decreasing the number of async_work_group_copy() calls that are used can decrease the average latency and improve the performance. For example, in kernel L100, only array u and array p need to be copied in, while only array cu and array h needs to be copied out. This is demonstrated in Figure 5.6. Therefore, kernel L100 and L200 can use four fewer async_work_group_copy() calls respectively. This leads to a speedup, which is more significant for L200, according to Diagram 6.12.

6.5.2. Iteration 8



Iteration 7* means using fewer async_work_group_copy()

Iteration 8* means using one out-of-order command queue to achieve the concurrent execution of kernels.

Iteration 8 means breaking down the periodic continuation kernels so they can be better parallelized

Diagram 6.14: Speedup of each Section of *Iteration 8* Compared to Baseline and *Iteration 7**

Kernel	Latency of *Iteration 7**	Kernel	Latency of *Iteration 8*	Latency of *Iteration 8**	Kernel	Latency of *Iteration 8***
L100	9145	L100_cu	7922	7923	L100_cu	7922
		L100_h	7912	7913	L100_h	7912
L200	9145	L200_p	7922	7923	L200_p	7922
		L200_u	7912	7913	L200_u	7912
L100_pc	8237	L100_pc	8237	8238	L100_pc_cu	2520
					L100_pc_h	5919
L200_pc	8237	L200_pc	8237	8238	L200_pc_p	5919
					L200_pc_u	2520

Iteration 7* means using fewer async_work_group_copy()

Iteration 8* means using one out-of-order command queue to achieve the concurrent execution of kernels.

Iteration 8 means breaking down the periodic continuation kernels so they can be better parallelized

Table 6.8: Average Latency Reported for each Kernel in *Iteration 7 and 8*.

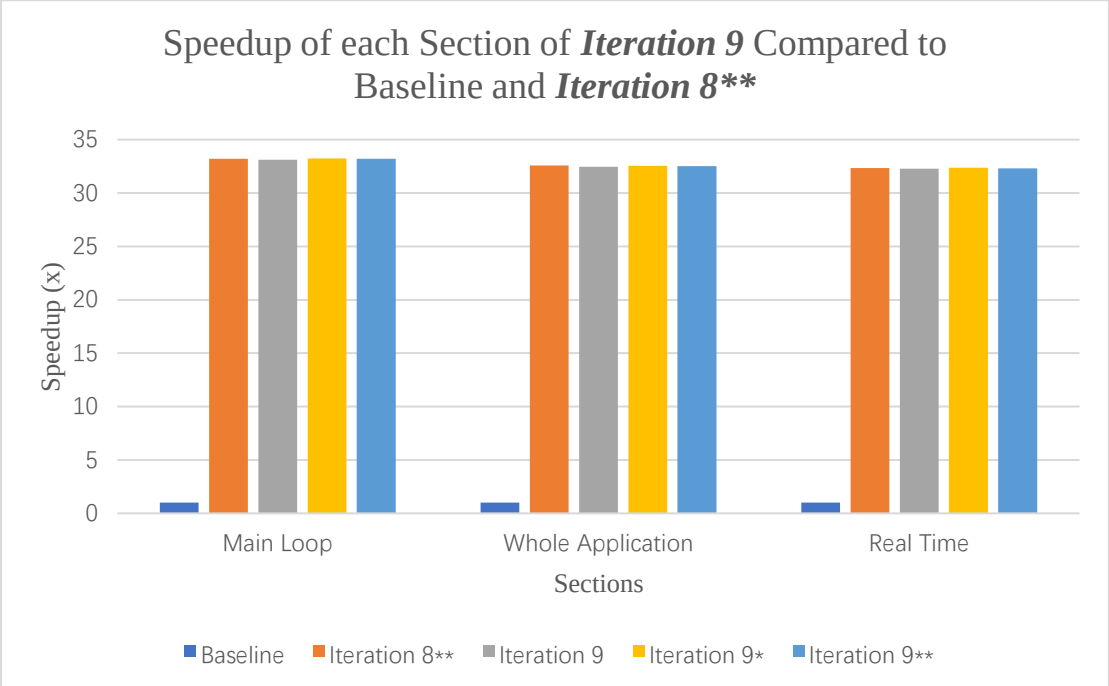
Iteration 8 aims to increase parallelism by executing the kernels concurrently, which is mentioned in Section 2.6.2. Diagram 6.14 demonstrates the speedup of the main loop, whole application and real time of baseline, ***iteration 7**** and ***iteration 8***. The latency of each kernel of ***iteration 7**** and ***iteration 8*** is presented in Table 6.5. It should be noted that in order to allow the concurrent execution of kernels, the execution time of kernel L100_cu, L100_h, L200_p, L200_u, L100_pc_cu, L100_pc_h, L200_pc_p and L200_pc_u are not measured explicitly, because the kernels need to be synchronized to obtain the correct execution time, while synchronization between kernels means kernels will be executed in a sequential way. Hence, in Diagram 6.14 only the speedups of main loop, whole application and real time are showed. Figure 5.10 provides a detailed description of this.

According to Diagram 6.14, the speedup of ***iteration 8*** in terms of main loop, whole application and real time is worsened by around 2.8% compared to the one in ***iteration 7****. This is because although the non-parallel code overhead is minimized, a scheduling overhead is introduced for executing kernels in two in-order command queues. It should also be noted that the speedup of ***iteration 8**** in terms of main loop, whole application

and real time is worsened by around 1.8% compared to the one in ***iteration 8.*** This suggest that it might be better to achieve the concurrent execution of kernels using multiple in-order command queues instead of one out-of-order command queue. ***Iteration 8***** achieved the best speedup by further breaking down the periodic continuation kernels and executing them in parallel. Figure 5.9 provides a description of this. Kernel L100_pc is divided into kernel L100_pc_cu and L100_pc_h respectively, the periodic continuation of array cu is conducted in kernel L100_pc_cu while the periodic continuation of array h is conducted in kernel L100_pc_h. The situation of kernel L200_pc is similar. Therefore, multiple in-order command queue will be used to achieve the concurrent execution of kernel in the future experiment.

As can be seen from Table 6.5, the latency reported during compilation of kernel L100_cu and L100_h is 7922 and 7912 respectively, since they are executed in parallel, the latency of kernel L100 is decreased from 9145 to 7922, according to Table 6.8. The situation of kernel L200 is similar. Furthermore, the latency of kernel L100_pc_cu and L100_pc_h is 2520 and 5919 respectively. Since they are now executed in parallel, the latency of kernel L100_pc is decreased from 8237 to 5919, according to Table 6.8. The situation of kernel L200_pc is similar. It is worth noting that, although the non-parallel code overhead in the periodic continuation kernels is minimized, a load imbalance overhead is then introduced. For example, as can be seen in Table 6.5, the latency of kernel L100_pc_cu and L100_pc_h is not the same. When they are executed in parallel, kernel L100_pc_cu needs to wait for kernel L100_pc_h to finish. The situation of kernel L200_pc_p and kernel L200_pc_u is similar.

6.5.3. Iteration 9



Iteration 8* means breaking down the periodic continuation kernels so they can be better parallelized

Iteration 9* means optimizing the code using only function pipelining.

Iteration 9* means optimizing the code using both function pipelining and function inline.

Diagram 6.15: Speedup of each Section of *Iteration 9* Compared to Baseline and *Iteration 8***

	Start Interval	Latency of ***Iteration 9***		Start Interval	Latency of ***Iteration 9****	Latency of ***Iteration 9*****
L100_cu	7924	7923	L100_cu	4249	5473	5473
L100_cu_calc	4248	4248	L100_cu _entry	0	0	0
			L100_cu _read	2451	2451	2451
			L100_cu _calc	4248	4248	4248
			L100_cu _write	1224	1224	1224

Iteration 9* means optimizing the code using only function pipelining.

Iteration 9** means optimizing the code using both function pipelining and function inline.

Table 6.9: Average Latency and Start Interval of Kernel L100_cu in *Iteration 9*

Optimization Iteration	Kernel	Start Interval	Average Latency
Baseline	L100	17190	17189
Iteration 1	L100	17190	17189
Iteration 2	L100	17190	17189
Iteration 3	L100	22623	22622
Iteration 4*	L100	16374	16373
Iteration 5*	L100	14042	14041
Iteration 6	L100	14042	14041
Iteration 7**	L100	9146	9145
Iteration 8***	L100_cu	7923	7922
Iteration 9****	L100_cu	4249	5473

*: Completely unroll the loops in each kernel.

**: Unroll the loops in kernel L100_pc and L200_pc using a factor of 2.

***: Using fewer async_work_group_copy() calls.

****: Using two in-order command queues along with eight kernels.

*****: Using both function pipelining and function inline.

Table 6.10: Start Interval and Average Latency of Kernel L100 and L100_cu of Baseline and *Iteration 1, 2, 3, 4, 5, 6, 7, 8 and 9*

Iteration 9 investigates the option of function inline and pipelining the function calls within a kernel, which is mentioned in Section 2.6.1. Diagram 6.15 demonstrate the speedup of the main loop, whole application and real time of baseline, ***iteration 8***** and ***iteration 9***. Table 6.6 present the start interval and latency information of kernel L100_cu in ***iteration 9, 9* and 9***** respectively, it helps explain how function pipelining works. It should be noted that ***iteration 9***, which is not optimized using either function pipelining or function inline, serves as a baseline here, it simply replaces all the original code with function calls. Figure 5.15 provides a description of how this is implemented in kernel L100_cu. Table 6.10 describes the start interval and average latency of some kernels of baseline and ***iteration 1, 2, 3, 4, 5, 6, 7, 8 and 9***. For simplicity, only the start interval and average latency of kernel L100 and L100_cu is demonstrated. The situation of other kernels is similar.

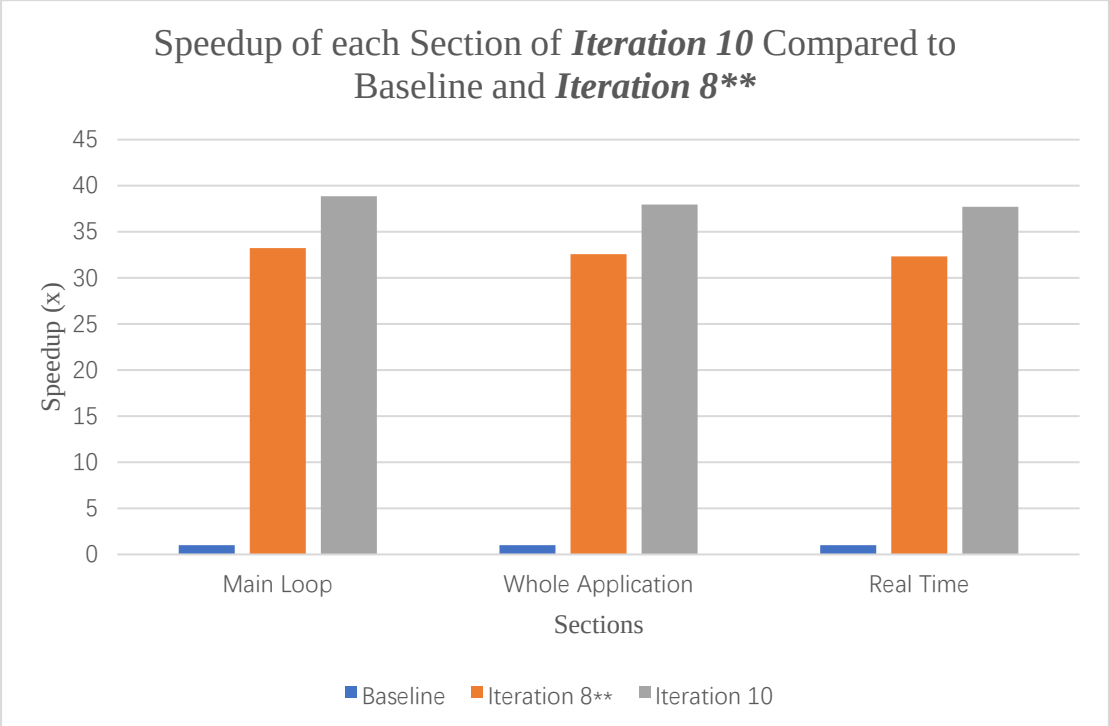
It should be noted that periodic continuation kernels cannot be optimized using function pipelining and function inline due to some compilation errors. What's more, the kernels cannot be optimized using only function inline because it triggers an "LLVM-link failed" compilation error. These issues are left for future investigation.

According to Diagram 6.15, there is no significant change in speedup after applying function pipelining or both function pipelining and function inline. This suggest either function pipelining or function inline is not working. However, some interesting facts can be obtained from Table 6.6. After replacing all the original code with function calls, the latency of kernel L100_cu is 7923, which is very close to the latency of kernel L100_cu in ***iteration 8***** (7922). The latency of the computation function (L100_cu_calc) of array cu is 4248. It is worth noting that the start interval is the same as the average latency now. After the function pipelining is applied, not only the latency of the whole kernel and the computation function (L100_cu_calc), but also the latency of the memory read function (L100_cu_read), memory write function (L100_cu_write) and the entry point (L100_cu_entry) is available. Furthermore, according to Table 6.10, for the first time among all the conducted experiments, it can be observed that the start interval of kernel L100_cu is significantly different from its average latency. Considering the fact that start interval is the minimum number of clock cycles that has to pass between the invocations of the compute unit for a given kernel, and average

latency is the number of clock cycles needed for the given kernel to finish execution, it can be concluded that the functions within kernel L100_cu are pipelined. Some more evidences, including the average latency of kernel L100_cu in ***iteration 9****, which is smaller than the one in iteration 9, suggests that the functions in kernel L100_cu are now not executed in a strict one-after-another way. What's more, the start interval still equals the average latency in the memory read function, array update function and memory read function, suggesting that it is the functions themselves rather than what is inside the function that are pipelined. However, there is still no change in latency after applying function inline, implying that it is not working. The situation in kernel L100_h, L200_p and L200_u is similar

It is interesting to observe that the speedup achieved suggests the function pipelining is not working while the latency information tells a completely opposite story. The reason behind this might be that "the absolute counts of cycles and latency are based on estimates identified during synthesis, especially with advanced transformations, such as pipelining and dataflow; these numbers might not accurately reflect the final results" according to ("SDAccel Environment Profiling and Optimization Guide," 2019). However, despite the failure in observing any significant improvement of speedup, function pipelining and function inline will still be kept and used in the future experiments.

6.5.4. Iteration 10



Iteration 8* means breaking down the periodic continuation kernels so they can be better parallelized

Diagram 6.16: Speedup of each Section of *Iteration 10* Compared to Baseline and *Iteration 8**

Iteration 10 investigates the option of merging array update kernel with periodic continuation kernel, which is mentioned in Section 2.6.1. Figure 5.17 provides a description regarding how this is implemented in kernel L100_cu. According to Diagram 6.16, the speedups of main loop, whole application and real time are improved by around 17%, due to the elimination of unnecessary data transfer. However, it should be noted that load imbalance overhead still exists due to the difference in latency of the periodic continuation operation of array cu and array h, as well as array u and array p, as mentioned in Table 6.8.

6.6. Comparison between Estimated Speedup and Achieved Speedup

In this section, the estimated speedup is obtained and compared against the achieved speedup. In order to acquire the estimated speedup, the estimated execution time needs to be calculated first, by utilizing the method introduced in Section 3.1.

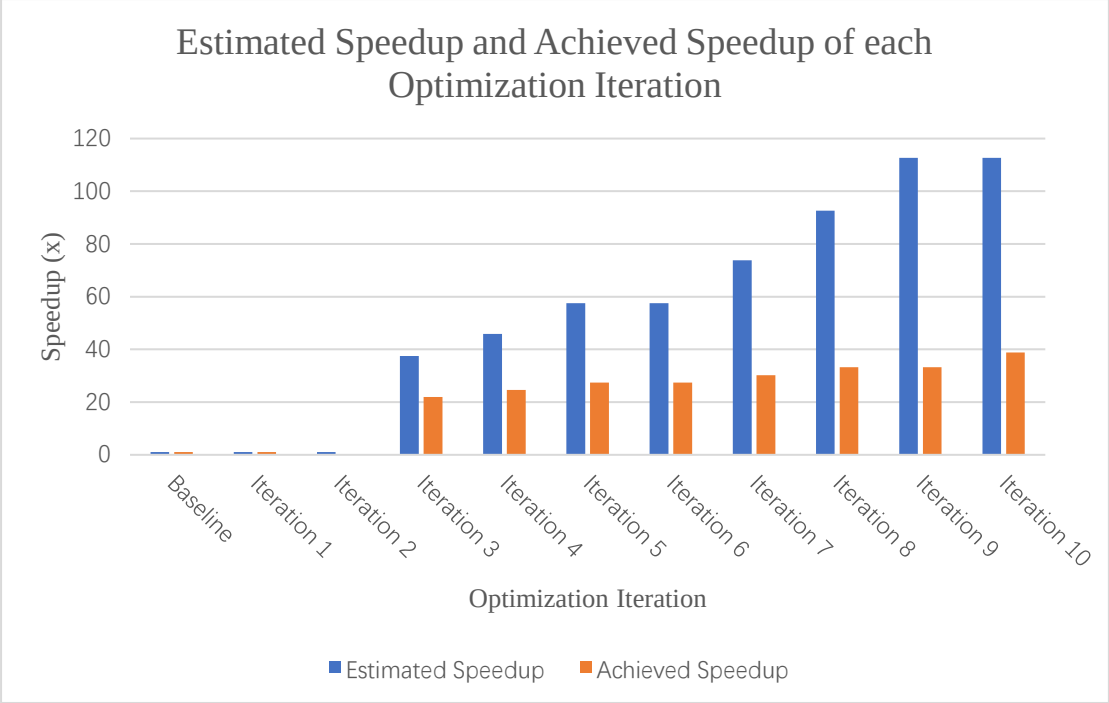


Diagram 6.17: Estimated Speedup and Achieved Speedup of each Optimization Iteration

The speedups of the main computation loop serve as the achieved speedups here, in order to make it comparable. Because only the latency of kernels can be obtained from the system estimate report, which means only the latency of the computation, but not the whole application, is reflected. According to Diagram 6.17, gaps between estimated speedup and achieved speedup start to appear since ***iteration 3***. The gaps suggest that there is still a significant amount of overhead needs to be minimized. The speedup gap becomes increasingly large as more and more optimizations are applied to the application, which implies that the use of optimization will introduce some other overheads. For example, execute kernels concurrently minimizes the non-parallel code overhead while introducing scheduling overhead and load imbalance overhead at the same time, as mentioned in Section 6.5.2. However, it should be noted that the latency information obtained from the system estimate report doesn't always accurately reflect what is really happening when the application is executed on FPGA, as mentioned in Section 6.5.3

6.7. Data with Bigger Problem Size

In this section, a larger problem size of 127 * 127 is applied to the simplified shallow water application to evaluate its performance. For simplicity, the larger problem size is only evaluated for baseline and ***iteration 10***, since ***iteration 10*** has the best speedup.

What's more, only the execution time of the main loop, whole application and real time is measured.

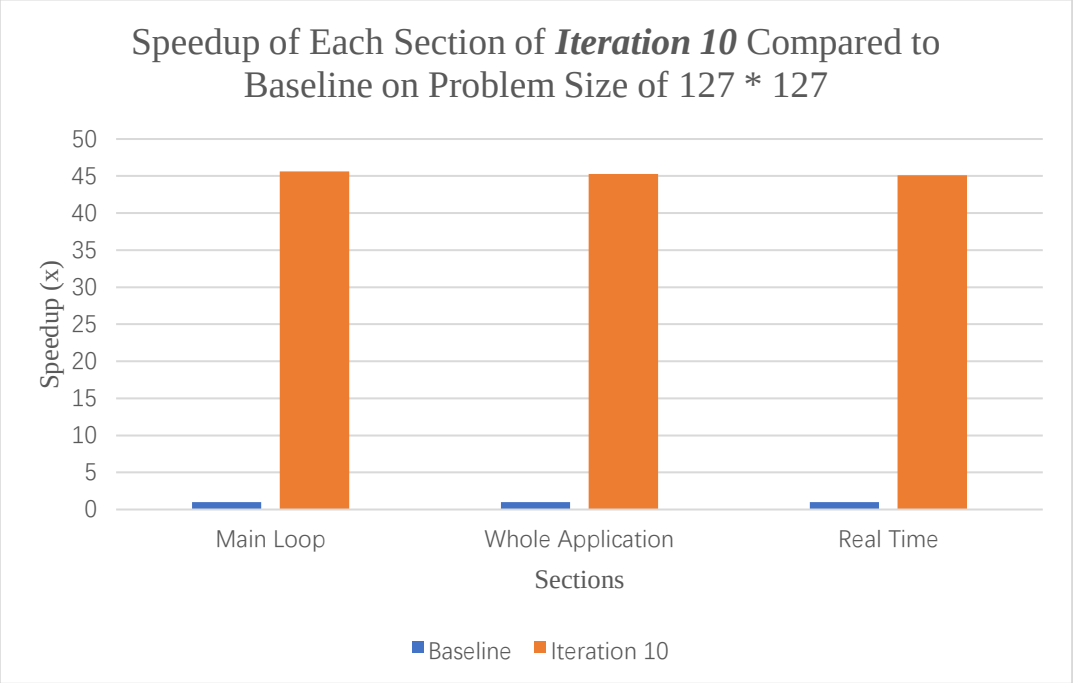


Diagram 6.18: Speedup of Each Section of *Iteration 10* Compared to Baseline on Problem Size 127 * 127

According to Diagram 6.18, a speedup of around 45x is achieved. By comparing Diagram 6.18 with Diagram 6.16, problem size 127 * 127 yields a better speedup. This phenomenon can probably be explained by Gustafson's Law, as more FFs, LUTs and BRAMs are used under problem size of 127 * 127. However, it should also be noted that it is a typical phenomenon that increasing problem size usually leads to better speedup. An in-depth analysis will be left for future work.

6.8. Performance Comparison among CPU, GPGPU and FPGA

In this section, the performance of ***iteration 10***, which is the simplified shallow water application that yields the best speedup so far, is compared against the performance of the simplified shallow water application that is implemented on CPU and GPGPU. The simplified shallow water application implemented on CPU is a sequential program developed using C, while the simplified shallow water application implemented on GPGPU is a program that is not highly optimized and developed using OpenCL. An Intel i7-6700 CPU and a Nvidia GT730 GPGPU is used in this section, their specifications are listed in Table 6.11. The performance of the simplified shallow water

application running on CPU is utilized as the baseline. For simplicity, only the execution time of main loop, whole application and real time are measured. Both problem size 65 * 65 and 127 * 127 are evaluated.

Specifications	Intel i7-6700 CPU	Nvidia GT730 GPGPU
Number of Cores	4	384
Base Clock Frequency	3.40 GHz	902 MHz
CPU Cache / GPGPU VRAM (MB)	8	2048
Memory Bandwidth (GB/s)	34.1	14.4
Thermal Design Power (W)	65	23

Table 6.11: Specifications of Intel i7-6700 CPU and Nvidia GT730 GPGPU

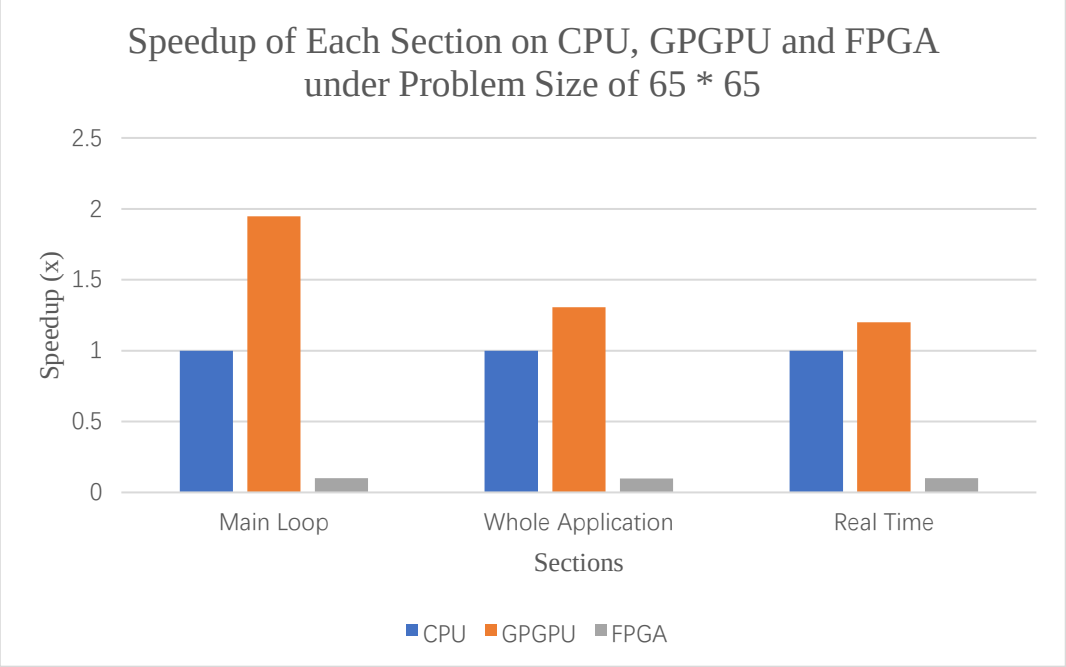


Diagram 6.19: Speedup of Each Section on CPU, GPGPU and FPGA under Problem Size 65 * 65

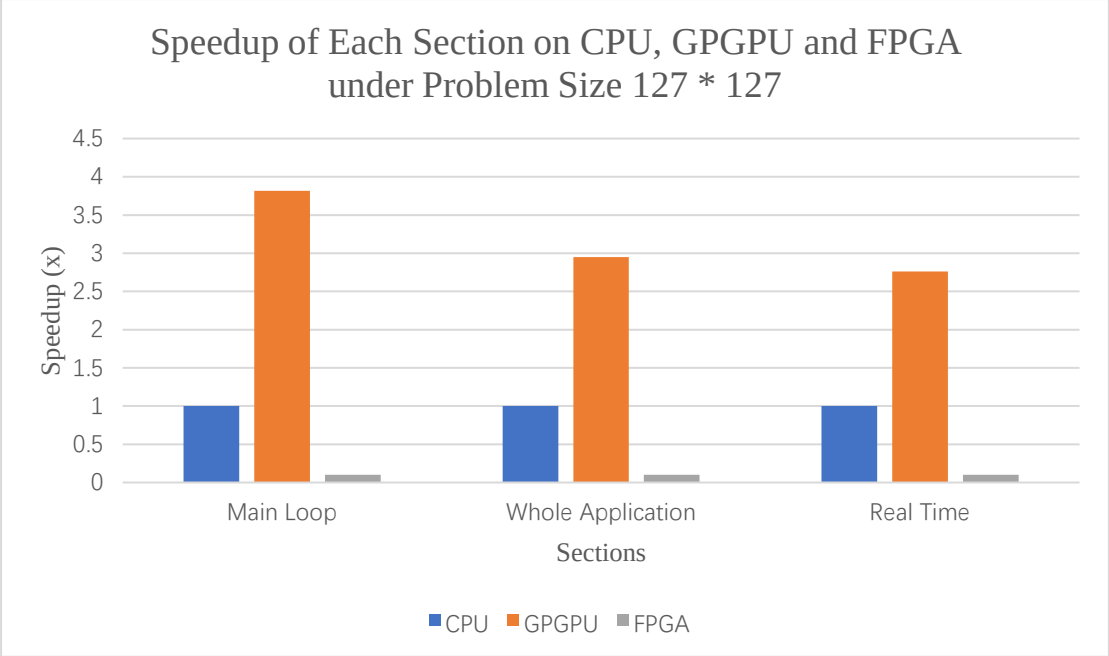


Diagram 6.20: Speedup of Each Section on CPU, GPGPU and FPGA under Problem Size 127 * 127

According to Diagram 6.19 and Diagram 6.20, the GPGPU yields the best speedup in terms of main loop, whole application and real time, under both problem size, while FPGA yields the worst. It should be noted that i7-6700 is a high-end Intel CPU while GT730 is an entry-level GPGPU, the Zynq® UltraScale+™ MPSoC ZCU102 board is a mid-range FPGA. What's more, the simplified shallow water application running on CPU is a sequential program, and the one running on GPGPU is not highly optimized either. However, the simplified shallow water application running on FPGA is a highly optimized one. In conclusion, if power consumption is not considered, GPGPU is still the best candidate for high-performance computing. However, it should be noted that if power consumption is considered, FPGA is still the most power efficient accelerator according to the work of (Targett et al., 2015), (Muslim et al., 2017), (Roozmeh and Lavagno, 2018), (Zohouri et al., 2016), (Nagasu et al., 2017), (Gan et al., 2013) and (Zhang et al., 2015).

The power consumption of FPGA is not obtained in this project due to the limited amount of time and the complexity of obtaining such data. Unlike CPU and GPGPU, the power of FPGA heavily depends on the hardware resource utilization, physical interface and design activity, for example clock frequency. Therefore, in order to acquire the power of FPGA, certain power monitoring software and hardware needs to be used in combination. The acquisition of the power consumption of FPGA will be left

for future work.

6.9. Summary

An explanation regarding how to interpret the latency and loop information provided in the system estimate report and HLS report has been presented in this chapter. A simple scalability model for exploring the relationship between the speedup and hardware resource utilization is developed. The reason why the emulator is unreliable is also explained in detail.

The experimental data analysis of this project is summarized in the following table. For simplicity, only the speedup of the main loop is presented here, along with throughput which is defined in Section 6.2.

Optimization Iteration	Speedup of Main Loop (x)	Throughput (Number of Elements per Second)	Number of IP Blocks
Baseline	1	558	14
Iteration 1	0.99989303	558	14
Iteration 2	N/A	N/A	N/A
Iteration 3	21.96232006	8759	10
Iteration 4*	24.60873861	2944	3
Iteration 5**	27.39488569	9833	9
Iteration 6	27.36177188	9821	9
Iteration 7**	30.22993021	10851	9
Iteration 8***	33.22599969	7951	6
Iteration 9*****	33.21620692	6624	5
Iteration 10	38.8561501	13947	9

*: Completely unroll the loops in each kernel.

**: Unroll the loops in kernel L100_pc and L200_pc using a factor of 2.

***: Using fewer async_work_group_copy() calls.

****: Using two in-order command queues along with eight kernels.

*****: Using both function pipelining and function inline.

Table 6.12: The Speedup of Main Loop and the Throughput of Baseline and *Iteration 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10*

According to Table 6.7, ***iteration 10*** yields the best speedup as well as throughput.

In conclusion, the non-parallel code overhead and memory access overhead are minimized by using loop pipelining, local memory, burst memory transfer, array partitioning, concurrent execution of kernels and kernel merging. However, other overheads also arise, including load imbalance overhead and scheduling overhead. These two overheads are all brought by the concurrent execution of kernels. Table 6.8 explains the load imbalance overhead seen between kernel L100_pc_cu and L100_pc_h, while Diagram 6.14 demonstrates the scheduling overhead of using two in-order command queues to execute kernels concurrently. The minimization of these overheads is left for future work.

The speedup of the simplified shallow water application is also obtained for a larger problem size of 127 * 127. The performance of ***iteration 10*** is compared against the performance of the simplified shallow water application executing on a modern CPU and GPGPU.

Chapter 7 Conclusion and Future work

This project implements and optimizes a simplified shallow water weather & climate forecasting application on an ARM CPU-FPGA heterogeneous system using OpenCL. A maximum speedup of the main loop of around 45x is achieved under problem size 127*127, compared with baseline. Background knowledge including the background of FPGA, OpenCL, stencil computation and a series of optimization methods are introduced. A detailed literature review is also conducted. The research methodologies necessary for this project, including performance estimation, overhead analysis, overhead minimization, execution time acquisition, the principle for applying optimization methods as well as a method for efficient FPGA programming are demonstrated. All the experiments conducted in this project, are explained in detail with the help of code snippets, based on several optimization iterations. The reason why the emulator is unreliable is discussed; a scalability model which aims to investigate the relationship between speedup and hardware resource utilization is introduced and demonstrated. A method of interpreting the latency and loop information obtained from the system estimate report and HLS report is explained. The experimental data are analysed using diagrams and tables that represent the speedup, average latency and hardware resource utilization. The overheads are also analysed and the reasons that lead to these overheads, as well as the reason why the overheads are minimized are explained. The performance of executing the highly optimized, simplified shallow water application on FPGA is compared with the performance of executing a not highly optimized simplified shallow water application on CPU and GPGPU.

7.1. List of Contributions

A series of contributions of this project is listed as follows:

- Implement and optimize a simplified shallow water application using OpenCL on an ARM CPU-FPGA heterogenous system, achieving a maximum speedup of the main loop of around 45x under problem size 127*127, compared with baseline.
- A method of efficient FPGA programming is discovered.
- The discovery of how to measure the execution time of kernel in a simple but correct and accurate way. The reason behind this is also understood.

- The discovery of the unreliability of SDx emulator, in terms of the measurement of execution time and the indication of performance improvement.
- The understanding of how to interpret the latency and loop information provided by the system estimate report and HLS report.
- The discovery of a simple scalability model which investigates the relationship between hardware resource utilization and speedup.

7.2. Future Work

Some possible future works are listed as follows:

Optimizing the simplified shallow water application using some other kernel optimization methods, which includes,

- Partitioning arrays in a block way.
- Manually vectorize the data using variable types like float2, float4, float8 and float16.
- Using memory object pipes for inter-kernel data transfer and data streaming.
- Using pragma “SDS data_zero_copy” for efficient data transfer between host memory and device memory.

Optimizing the simplified shallow water host code using another OpenCL API execution model named clEnqueueTask.

Optimizing the simplified shallow water application using SDx-related optimizations, which includes:

- The number of compute units of each kernel.
- The data motion network clock frequency.
- The port data width of each kernel.
- Using dedicated memory ports for each global array.

Other future work includes the optimizations on the computable solution level, which includes:

- Using 2D-NDRange kernel.
- Initializing array u and array p on FPGA.

- Stop conducting periodic continuation since they are prepared for systems with cache.
- Overlapping the execution of kernels.
- Optimizing the simplified shallow water application on multiple FPGA.

Bibliography

Application programming interface. (2019). [Online]. Available at: https://en.wikipedia.org/wiki/Application_programming_interface (Accessed 1 Sep. 2019).

Cong, J., Fang, Z., Hao, Y., Wei, P., Yu, C.H., Zhang, C., Zhou, P., 2018. *Best-Effort FPGA Programming: A Few Steps Can Go a Long Way*. arXiv:1807.01340.

Conte, A. (2019). *FPGA based acceleration of a particle simulation High performance computing application*. Master. Politecnico di Torino.

da Silva, B., Braeken, A., D'Hollander, E.H., Touhafi, A., 2013. *Performance Modeling for FPGAs: Extending the Roofline Model with High-Level Synthesis Tools*. International Journal of Reconfigurable Computing. https://doi.org/10.1155/2013/428078

Dohi, K., Fukumoto, K., Shibata, Y., Oguri, K., 2013. *Performance modeling and optimization of 3-D stencil computation on a stream-based FPGA accelerator*, in: 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig). Presented at the 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp. 1–6. https://doi.org/10.1109/ReConFig.2013.6732318

Düben, P.D., 2018. *A New Number Format for Ensemble Simulations. Journal of Advances in Modeling Earth Systems* Volume 10, pp. 2983–2991. https://doi.org/10.1029/2018MS001420

Fifield, J., Keryell, R., Ratigner, H., Styles, H., Wu, J., 2016. *Optimizing OpenCL applications on Xilinx FPGA*, in: Proceedings of the 4th International Workshop on OpenCL - IWOCL '16. Presented at the the 4th International Workshop, ACM Press, Vienna, Austria, pp. 1–2. https://doi.org/10.1145/2909437.2909447

Gan, L., Fu, H., Luk, W., Yang, C., Xue, W., Huang, X., Zhang, Y., Yang, G., 2013. *Accelerating solvers for global atmospheric equations through mixed-precision data flow engine*, in: 2013 23rd International Conference on Field Programmable Logic and Applications. Presented at the 2013 23rd International Conference on Field programmable Logic and Applications, pp. 1–6. https://doi.org/10.1109/FPL.2013.6645508

Georgopoulos, K., Mavroidis, I., Lavagno, L., Papaefstathiou, I., Bakanov, K., 2019. *Energy-Efficient Heterogeneous Computing at exaSCALE—ECOSCALE*, in: Kachris, C., Falsafi, B., Soudris, D. (Eds.), Hardware Accelerators in Data Centers. Springer International Publishing, Cham, pp. 199–213. https://doi.org/10.1007/978-3-319-92792-3_11

Gorlani, P. (2017). *FPGA in HPC: High Level Synthesys of OpenCL kernels for Molecular Dynamics*. Master. Scuola Internazionale Superiore di Studi Avanzati.

MaxelerOS | Maxeler Technologies, n.d. URL https://www.maxeler.com/products/software/maxeleros/ (accessed 8.14.19).

MaxGenFD | Maxeler Technologies, n.d. URL https://www.maxeler.com/products/software/maxgenfd/ (accessed 8.14.19).

Mondigo, A., Ueno, T., Sano, K., Takizawa, H., 2019. *Scalability Analysis of Deeply Pipelined Tsunami Simulation with Multiple FPGAs*. IEICE Trans. Inf. & Syst. E102.D, pp. 1029–1036. https://doi.org/10.1587/transinf.2018RCP0007

Muslim, F.B., Ma, L., Roozmeh, M., Lavagno, L., 2017. *Efficient FPGA*

Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. IEEE Access Volume 5, pp. 2747–2762. https://doi.org/10.1109/ACCESS.2017.2671881
Nagasu, K., Sano, K., Kono, F., Nakasato, N., 2017. *FPGA-based tsunami simulation: Performance comparison with GPUs, and roofline model for scalability analysis*. Journal of Parallel and Distributed Computing Volume 106, pp. 153–169. https://doi.org/10.1016/j.jpdc.2016.12.015
OpenCL. (2019). [Online]. Available at: https://en.wikipedia.org/wiki/OpenCL (Accessed: 1 April 2019).
Pappas, M. (2012). *Parallelisation of Shallow Water Simulation For Heterogenous Architectures*. MSc. The University of Manchester.
Parker, M., 2017. *Understanding Peak Floating-Point Performance Claims*. Technical report (white paper): Intel, WP-01222-1.1
Riley, G.D., Bull, J.M., Gurd, J.R., 1997. *Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics*, in: Proceedings of the 11th International Conference on Supercomputing, ICS '97. ACM, New York, NY, USA, pp. 36–43. https://doi.org/10.1145/263580.263589
Roozmeh, M., Lavagno, L., 2018. *Design space exploration of multi-core RTL via high level synthesis from OpenCL models*. Microprocessors and Microsystems Volume 63, pp. 199–208. https://doi.org/10.1016/j.micpro.2018.09.009
Sano, K., Hatsuda, Y., Yamamoto, S., 2014. *Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth*. IEEE Transactions on Parallel and Distributed Systems Volume 25, pp. 695–705. https://doi.org/10.1109/TPDS.2013.51
"SDAccel Environment Profiling and Optimization Guide", UG1207, Xilinx, 2019.
"SDSoC Profiling and Optimization Guide", UG1235, Xilinx, 2019.
"SDx Pragma Reference Guide", UG1253, Xilinx, 2019.
Shallow water equations. (2019). [Online]. Available at: https://en.wikipedia.org/wiki/Shallow_water_equations (Accessed: 1 April 2019).
Skalicky, S., López, S., Łukowiak, M., Letendre, J., Ryan, M., 2013. *Performance Modeling of Pipelined Linear Algebra Architectures on FPGAs*, in: Brisk, P., de Figueiredo Coutinho, J.G., Diniz, P.C. (Eds.), Reconfigurable Computing: Architectures, Tools and Applications, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 146–153.
Strenski, D., Simkins, J., Walke, R., Wittig, R., 2008. *Evaluating FPGAs for floating-point performance*, in: 2008 Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications. Presented at the 2008 Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications, pp. 1–6. https://doi.org/10.1109/HPRCTA.2008.4745680
Targett, J.S., Niu, X., Russell, F., Luk, W., Jeffress, S., Düben, P., 2015. *Lower precision for higher accuracy: Precision and resolution exploration for shallow water equations*, in: 2015 International Conference on Field Programmable Technology (FPT). Presented at the 2015 International Conference on Field Programmable Technology (FPT), pp. 208–211. https://doi.org/10.1109/FPT.2015.7393152
The OmpSs Programming Model | Programming Models @ BSC, n.d. URL https://pm.bsc.es/ompss (accessed 8.14.19).
VHDL. (2019). [Online]. Available at: https://en.wikipedia.org/wiki/VHDL (Accessed

1 Sep. 2019).
Vivado High-Level Synthesis. (2019). [Online]. Available at: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html (Accessed 1 Sep. 2019).
Waidyasooriya, H.M., Hariyama, M., Uchiyama, K., 2018. *Design of FPGA-Based Computing Systems with OpenCL*. Springer International Publishing. https://doi.org/10.1007/978-3-319-68161-0
Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J., 2015. *Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks*, in: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '15. ACM, New York, NY, USA, pp. 161–170. https://doi.org/10.1145/2684746.2689060
Zohouri, H.R., Maruyama, N., Smith, A., Matsuda, M., Matsuoka, S., 2016. *Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs*, in: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Presented at the SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 409–420. https://doi.org/10.1109/SC.2016.34