Exploring High-Level Synthesis to FPGA-based Highly Efficient Stencil Computation

March 2021

Changdao Du

Exploring High-Level Synthesis to FPGA-based Highly Efficient Stencil Computation

Graduate School of Systems and Information Engineering

University of Tsukuba

March 2021

Changdao Du

Abstract

Many high-performance computing (HPC) applications, such as training neural networks, image processing, and weather simulation, have continuously increasing requirements for the computing capabilities. Traditional CPU-based computing platforms often can not meet these requirements due to unsustainable performance gains of CPUs. Therefore, to solve that problem, the use of dedicated hardware accelerators to improve the performance of these HPC applications is attracting attention in the last few years. GPUs have already been proven to be the most popular hardware accelerators in the past decade. This is mainly due to their parallel many-core architecture and high-speed storage bandwidth. However, these devices also suffer from the strong needs for the power supply and limited I/O interfaces.

Recently, many studies have tried to use FPGAs as the dedicated accelerators to deal with these HPC applications. The results demonstrate that FPGAs also have the potential to provide GPU-level computing power and maintain energy efficiency. However, FPGAs' hardware-based design flow usually hinders the way to popular them to mainstream users in the HPC field. Although both academia and industry have developed high-level synthesis (HLS) tools that allow users to directly program FPGA with conventional languages, e.g., C or C++, to improve usability and productivity. Programming on FPGAs with these HLS tools to achieve high efficiency and good performance is still a time-consuming task, and lack of knowledge about optimization strategies and techniques may lead to poor scalability and portability. Therefore, in this thesis paper, I discus the corresponding optimization strategies and techniques to use HLS developing method for HPC applications, specifically, for the stencil computations. Due to the low arithmetic intensity and irregular memory access pattern, the peak performance of fixed architecture, e.g., GPUs or CPUs often can not be reached for computing stencil kernels.

In the first part of this thesis, I set two typical computational fluid dynamic (CFD) simulation modes, i.e., lattice Boltzmann method (LBM) and lattice gas cellular automata (LGCA) as the target applications. The proposed architecture design can take advantage of both spacial and temporal parallelism to increase the simulations performance. During the implementation process, I adopt bunch of HLS optimization strategies. In addition, I also discuss the design portability issue that related to specific HLS developing environment. I evaluate the architecture on a Xilinx VCU1525 FPGA

board with the SDAccel HLS developing environment. The evaluation results show that the simulation performance can scale well with the two main design parameters-i.e., for spatial and temporal domain. For the LGCA simulation, the best result of VCU1525 FPGA achieves 17130 MLUPS, which is 30 times faster than a i7-6700 CPU-based implementation and 3 times faster than a Quadro P5000 GPU-based implementation. For the LBM simulation, the result achieve the 4919 MLUPS, which is a competitive result compared to a GTX Titan GPU implementation.

In the next part, according to the previous work, I generalize the proposed architecture to normal stencil applications by using 3 benchmarks, Sobel filter, Laplace equation, and Himeno benchmark. Since an HBM-connected FPGA board is used, I am able to explore the complete design space for using spatial parallelism. The exploration process provide an opportunity to utilize the computation reusability inside the certain stencil kernels. I evaluate the architecture on a Xilinx Alveo U280 board. For non-iterative stencil benchmark, e.g., Sobel 2D, the results show the architecture can achieve 10x-20x higher performance than traditional FPGA boards. This mainly due to the advantage of HBM memory bandwidth. The resource consumption report shows that by reusing the calculation results, the cost of the LUTs can be reduced by about 20%.For the 2D Laplace equation, compared with previous design approach, I use larger value of spatial parallelism parameter to scale-up the application performance. The advantage of using the large value of spatial parameter enables users to share some FPGA hardware resources like BRAMs inside space domain. This situation is more serious in the 3D stencil applications, e.g., Himeno. Since the custom buffer needs to buffer more data for 3D stencil kernel. The results show that the maximum performance of Alveo U280 with large spatial parallelism can achieve 4x higher performance compared with the version with large temporal parallelism in the Himeno benchmark.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the past half-century, the CPUs have been the main component of building a high-performance computing (HPC) platform. However, due to the slowdown of Moore's Law [1], the performance of CPUs has not shown a dramatic increase in recent years. Also, the power consumption of the single-chip CPUs are no longer benefits from the size of MOSFET transistors (Dennard Scaling) as before [2]. For these reasons, the CPU-based computing platforms often can not provide the sustainable computing capabilities and energy efficient in many HPC applications, such as training the training neural networks, performing hydrodynamic simulations, or searching data engines. Therefore, building modern computing platforms with pure CPUs seems not a promising solution.

For the last decade, Graphics Processing Units (GPUs) which is originally design for rendering images in the computing system has been proven to be also good at dealing with high performance computing tasks [3–5]. This is mainly due to the high parallel architectures, i.e., large number of cores, fast external memory, and native supported floating-point processing abilities. Compared to CPUs, the computing platforms using GPUs as dedicated hardware accelerators usually can achieve order of magnitude performance improvement in high-performance applications. However, the power consumption of GPUs also cannot be ignored, e.g. a single GPU often require 100 W to 300 W to run in full frequency and the corresponding cooling system also consume large amount of power.

Recently, researchers also found Field-Programmable Gate Arrays (FPGAs), which are prefabricated semiconductor devices that users can implement or re-implement any digital logic on them, show great potentials in HPC area [6–10]. Compared to GPUs or CPUs, although FPGAs' work frequency is low, FPGA-based platforms can explore the fine-grained parallelism of target applications by constructing application-specific pipeline systems. Unlike the pipeline system on fixed architecture, e.g., CPU, the custom pipeline system on FPGA do not need to process extra stages like fetch or decode, they can be deeply optimized only for the target applications. Moreover, the parallel structure of large numbers of logic cells and dedicated on-chip DSPs allow users to build a fully customize design. As a result, These FPGA-based platforms have the possibilities to provide GPU-like performance results and maintain lower energy costs in various HPC applications. Besides, the configurable I/O also helps FPGAs to be integrated in complex computing systems. These features make FPGAs gain a lot of attentions as a suitable hardware accelerator. For example, after Intel acquired Altera in 2015, in 2020 AMD also acquired Xilinx, another FPGA leader company, to improve the competitiveness in the HPC field. Catapult, a Microsoft research project, already used FPGAs as their cloud computing accelerators [11]. They can achieve 21 cents per million pictures in terms of cost for a deep neural network (DNN) application with ResNet-50 by using the Arria 10 FPGAs.

1.2 Motivation and Contribution

To obtain the expected performance of target applications on FPGAs, the developing methods play a significant role. The traditional developing method of FPGAs is based on the register transfer level (RTL) description which is often programmed by hardware-oriented languages e.g., VHDL or Verilog. However, designers using the hardware-oriented languages are often required to spend considerable time on logic details rather than optimizing the key area implementation in target application. Moreover, to proper describe the RTL design with such languages like VHDL, it also need designers have strong background in Integrated Circuit (IC) design e.g., data paths or/and Finite State Machine (FSM). As a result, the RTL-based developing method hurts the design productivity of FPGAs and block the way to introduce FPGAs to the mainstream software-based users of HPC fields.

To improve productivity and usability of FPGAs, high-level synthesis (HLS) developing method starts to gain their popularity in recent years [12]. Both academia and industry have developed various HLS tools, e.g., Vivado HLS [13], Intel HLS SDK [14], or LegUp [15], to help users to generate the RTL level design directly from the high abstraction description that programmed by common software-based language, such as, C, C++, or OpenCL.

In this thesis paper, I focus on presenting an HLS-based FPGA design for stencil applications. Stencil computations are wildly used kernels in many HPC applications, such as hydrodynamic simulations, solution of mathematical equations, and image processing technologies. However, since stencil computations generally do not have strong computing intensive and the memory access patterns are irregular, the computing platforms such as GPUs or CPUs often can not utilize their maximum computing capabilities. For example, research [16] shows only about 20% computing power of CPUs is delivered for implementing some CFD simulations. [17] demonstrates no more than half of peak performance of GPUs can be used for general stencil computations, since their fixed memory architecture and external memory bandwidth limit the use of computing power.

For various stencil applications, obtaining enough high performance on the target computing devices is always the most significant goal. Many performance optimizing techniques and strategies targeting for using the high-level languages such as C, or C++ to implement the stencil computations on CPUs or GPUs are well discussed. Nonetheless, for using these languages on FPGAs, especially by the HLS methods, still few principles or rules have been established for performance optimization. Simply reusing the CPU-based or GPU-based software code in HLS compilers often causes performance degradation. In the worst case, the HLS compiler may generate the incorrect RTL structure or fail at the implementation stage. This mainly due to the following reasons:

- High-Level languages, e.g., C are designed for developing program on CPUs, not for the digital logic structure. The traditional software development techniques e.g., pointer arithmetic, dynamic memory location, or recursive function call often do not work well on FPGAs.
- The hardware components inside FPGA such as FIFOs, BRAMs are not originally used in these languages. Designing efficient FPGA structures requires the flexible use of FPGA hardware components. For example, building the cache system by using on-chip memory resource BRAMs can significantly reduce the FPGA memory bandwidth bottleneck.
- To achieve high performance, the HLS tool allows developers to explore large design spaces with different choices of optimizations strategies. Choosing the optimal combinations of these strategies remains a time-consuming task.

Therefore, for efficiently implementing the stencil architecture on FPGAs by exploring the HLS developing method, the developers need to perform optimizations that can help FPGAs generate the proper hardware structure with the HLS design. These optimization strategies and techniques are intended to cost the optimal on-chip computing and memory resources to build a application specific architecture on the target FPGA. Compared to using the high-level languages on fixed architectures like CPUs, the optimizations objectives to increase the stencil computation performance for adopting HLS on FPGAs can be characterized as the follows 3 parts:

1. Building pipeline system with the minimum value of initial interval (II) to benefit from fine grained parallelism. Although HLS tools can use directive *pragmas* such as *pragma pipeline* to automatically pipeline the target design code, due to FPGA hardware limitations, some CPU-based code may not achieve full pipeline (II = 1). Users should explicitly identify the reasons and optimize the corresponding code, e.g., optimizing data dependency issue among loops to reduce the II;

2. To exploit coarse grained parallelism, the vectorization and duplication process can be solved with manually adding the identical computing functions and/or using *pragma loop unroll* to fold the loops insides target code. For CPU-based users, these operations are similar concept as multi-thread programming. However, for using HLS on FPGA, the users also need to make extra optimizations of the FPGA structure, such as re-partitioning the on-chip memory resources.

3. Re-organization of data access pattern to optimize external memory bandwidth utilization, e.g., using FIFO structure to build the stream data-flow interfaces between the functions. Compared to CPUs, due to lack of an explicit cache system, the data movement from external memory and on-chip memory should be carefully optimized.

To achieve these 3 optimization objectives, there still exist many challenges, such as, the data dependency between the loops can stall the pipeline engine of the HLS compiler, the default vectorization pragma *pragma loop unroll* may use redundant memory resources to feed the data bandwidth requirement by the computing logic, and the memory access efficiency of the bus interface needs to be specially configured. In this thesis, I will introduce a bunch of HLS specific optimization strategies for code conversion from fixed architecture (e.g., CPU) to FPGAs with the various stencil applications and I also discuss how these strategies affect the performance of target stencil applications. In addition, I also show the HLS potability problem and give some hints to help developers realize which parts of the code might compromise the portability of the HLS design during the implementation of target stencil applications. The main contributions are:

Computational Fluid Dynamics

For various Stencil applications, the thesis paper uses the Computational Fluid Dynamics (CFD) simulations as the initial study target benchmarks. CFD analyze fluid dynamics by numerically solving the equations of particles motion and it has been wildly used in many scientific and engineering areas. I propose an HLS-based architecture for implementing CFD simulations on FPGA. I select two typical CFD modes, i.e., Lattice Boltzmann Method (LBM) and Lattice Gas Cellular Automata (LGCA). The main contributions of this work include following parts:

- I propose a HLS-based architecture design that can exploit parallelism of CFD simulations in both spatial and temporal domains. The key to achieve high performance in CFD simulations is to run the simulations as much parallel as possible. The two kinds of parallelism are characterized as two main design parameters. The proposed architecture can scale-up performance by using both two parameters based on the target FPGA resources limitations.

- I introduce a custom buffer design that explicitly uses FIFOs and registers to solve the data dependency issue on the temporal domain. Compared to studies adopting shift-register based buffer design under the Intel HLS developing environment, the proposed custom buffer design can work well under the different HLS platforms.
- For the LGCA application, I propose a object-based vectorization method to use the spatial parallelism. This vectorization utilize the arbitrary integer datatype to implicitly increase the spatial parallelism value. By using this method, users do not need to explicitly change the on-chip buffer structure to feed the vectorized simulation processing units. For the LBM application, I propose a scalable version of FIFO-based custom buffer. This buffer can exploit the data locality of target stencil kernel to achieve the full data reuse ability with the optimal memory resource cost.
- A performance model is devised to tune the design parameters. To obtain the maximum performance of CFD simulation on the target FPGA board, the users need to search an optimal combination of design parameters, especially for spatial parallelism and temporal parallelism parameters.

General Stencil Computations with HBM

According to the HLS-based CFD simulation framework, I extend the existing architecture to the general stencil applications. I choose 3 typical benchmarks. They are Sobel 2D filter stands for image processing, Laplace equation (4-point) for mathematical equation, and Himeno for 3D hydrodynamic simulation. Besides, in this study, I use the state-of-the-art FPGA board Xilinx U280. Compared to traditional FPGAs, the board U280 is equipped with High Bandwidth Memory (HBM) banks, which can increase the FPGA external memory to 460 GB/s. This huge advantage creates new design possibilities in the FPGA architecture. Based on my previous work, the major contribution in this study are as follows:

- Traditional FPGAs often use DRAM banks, e.g., DDR3 or DDR4 as external memory, which limits the design space exploration for using the spatial parallelism. With the help of HBM memory, I propose a FPGA-based stencil accelerator design architecture which can significantly extend the design possibilities in the spatial domain.
- To fully explore the design possibilities in spatial domain, I generalize the custom buffer design that can scale parallelism value in multiple space dimensions. By considering the portability of the design, the proposed method insist on using the conventional FPGA resource or IP cores, e.g., FIFOs and registers to clearly

describe the data movement inside the custom buffer design. For stencil kernels with computation reuse-ability, the proposed custom buffer provide a design option to utilize this optimization opportunity.

- I introduce an optimization flow for efficiently using the HBM memories. Compared with FPGAs, the GPUs have a highly efficient dedicated memory control module. This module can implicitly merge the memory accesses to fully utilize external memory of GPUs. For FPGAs, it relies on the users to control the memory access pattern. Based on the HLS developing method, I first build a data distribution and collection buffer to increase the data width of the FPGA memory interface. Secondly, I utilize the burst mode of the AXI bus as much as possible to achieve higher transmission efficiency.

1.3 Thesis Outline

The rests of this thesis is organized as follows.

In Chapter 2, I introduce the fundamental knowledge of FPGA architecture. The FPGA design flow is also shown in this chapter. At last, I present two typical FPGA developing methods.

In Chapter 3, I present the previous work of how to implement the stencil architecture on FPGA with the HLS developing method. Then, I specifically describe the remaining challenges.

In Chapter 4, I first explain the basic concept of CFD, a and demonstrate two specific models of CFD, i.e., LGCA and LBM. Then, I use these 2 typical stencil application to describe the key optimization approaches to improve the simulation performance on FPGA with HLS method. After that, I describe detail implementation process of the simulation. At last, I discuss the results and compare them with other platforms.

In Chapter 5, I extend the architecture in chapter 2 to fully explore the design space for stencil applications. Since I use a HBM-enable FPGA board, I briefly show the structure of HBM in the beginning of this chapter. Next, I mainly discuss how to modify out stencil architecture for expanding the design space with the support of HBM. By fully exploring the design space, I find the extra optimization chances for re-using the computation results of certain stencil kernels. I use 3 typical benchmarks to verify the proposed structure.

In Chapter 6, I present the conclusion and future work of this thesis paper.

Chapter 2

FPGA Architecture and Design Method

In this chapter I first introduce the fundamental knowledge of general FPGA architecture, and show the basic logic elements inside an FPGA. Compared to fix architecture such as CPUs, I explain that how FPGAs use these elements to realize the digital logic operations. Then, I describe the work flow for developing FPGA design. In this section, I specifically talk about the HLS method which uses conventional languages, e.g., C or C++ to describe the FPGA design.

2.1 Field-Programmable Gate Array

Field-programmable gate array (FPGA) is a type of prefabricated semiconductor integrated circuit (IC) device on which users can implement or re-implement any digital logic functions after fabrication. Compared with traditional IC devices, the most advantage of FPGAs is their logic and connections can be programmed like software code on a CPU. The history of FPGA can be traced back to 1985 when the first FPGA device-i.e. XC2064, was invented by the creators of Xilinx [18]. Compared with XC2064 which only owns 64 of logic gate blocks, today's FPGAs already have the capabilities to hold millions logic gates and thousands of dedicates DSPs and BRAMs blocks. These tremendous advances and the re-configurable feature allow FPGAs to be adopted in many different markets, such as aerospace instruments, automotive systems, and HPC fields. In this thesis, I mainly discuss how to use FPGA to achieve enough high performance for the stencil applications.

2.1.1 FPGA Architecture

The overview structure of an FPGA chip can be shown in Figure 2.1. An FPGA chip mainly includes following 4 types of elements-i.e., configurable logic blocks (CLBs), DSPs, BRAMs, and wires. Unlike old type FPGA chips which mainly use the CLBs to perform float-point computations, modern FPGAs have already integrated with dedicated DSPs, allowing FPGA to provide similar computing capability as GPUs. Besides, the large amount of on-chip configurable block memories (BRAMs) are also integrated inside modern FPGAs. By connecting these logic elements with wires, developers have the flexibility to efficiently implement target applications on FPGAs. The details of these elements are shown in the following.

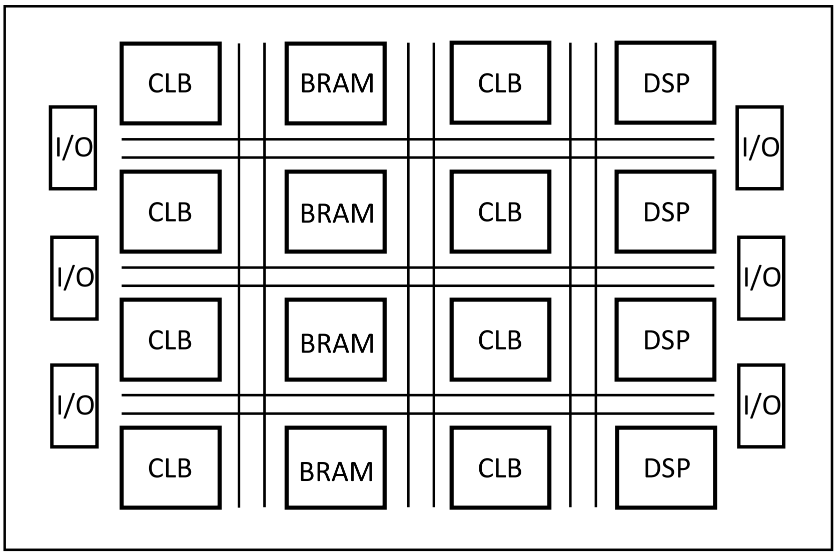


Figure 2.1: The overview of FPGA basic architecture

Configurable Logic Block

As I stated above, CLBs are the fundamental logic cells of FPGAs, allowing developers to configure them for implementing basic logic operations. When multiple CLBs are linked together with routing resources, they also can execute complex logical functions. An example of a CLB structure is shown in Figure 2.2. A typical CLB contains small components such as look-up tables (LUTs), flip-flops (FFs), and multiplexers.

- Look-up tables: The LUT stores a predefined truth table of outputs value list in which different combinations of n inputs can generate any logic functions of 2^n. In another word, the LUT actually emulates the logic operation gates rather than directly calculate the results. For example, the "AND" logic gate with two inputs "in0" and "in1" can be implemented by using the truth table value "0001" in the Figure 2.2.

- Flip-flop: The FFs are the minimum storage unit for FPGAs. Each FF is a binary register used to store the logical state with the clock signal of FPGA. Generally, the FFs are in company with the LUTs. The clock signal is controlled by the clock enable pin. The FF can latch the result of LUT. When the clock enable is set to 1 and the positive edge of clock wave is coming, FF can transfer the latched data to the output.

- Multiplexer: The multiplexer is used to select the inputs from the LUT and the FF, and pass the selected data to the output of CLB. It can improve the design flexibility of CLB.

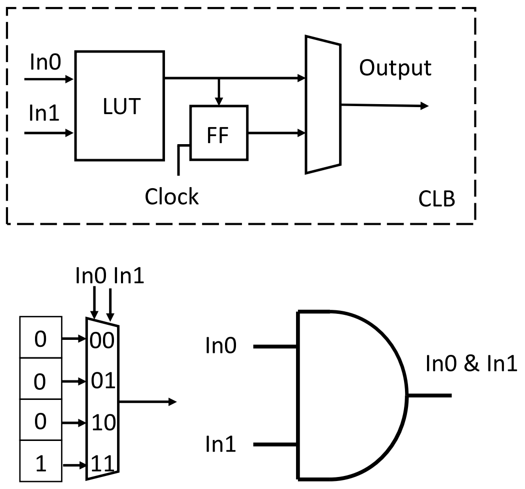


Figure 2.2: The example of CLB structure

Digital Signal Processing Block

The on-chip DSP blocks are complex calculation elements inside FPGAs. One example of Xilinx DSP block is shown in Figure 2.3 [19]. Inside the DSP block, there are 3 main computing modules, i.e, add/sub module, multiply module and the add/sub/accumulate module. They form a chain structure to implement some independent math functions, such as multiplication, multiply-accumulate operation, or pattern detecting. Although these functions also can be realized by using CLBs, implementing them with dedicated DSP block will improve the design efficiency, save energy consumption, and optimize the FPGA operating frequency.

Block Random Access Memory

As using DSP blocks for arithmetic computing, the on-chip BRAMs are dedicated storage elements of FPGA [20]. Although LUTs also can store some data by using the truth tables, their storage capacities are very limited. Compared to LUT storage (also known as distributed memories), a typical BRAM block often can store 18k or 36k bits in one device and state-of-art FPGAs often have thousands of these BRAMs, which allows FPGAs to hold dozens of megabytes data inside the chip. The connections of these BRAMs also can be fully customized by the developers. As a result, developers often use these BRAMs to build a high efficiency application-specific cache structure

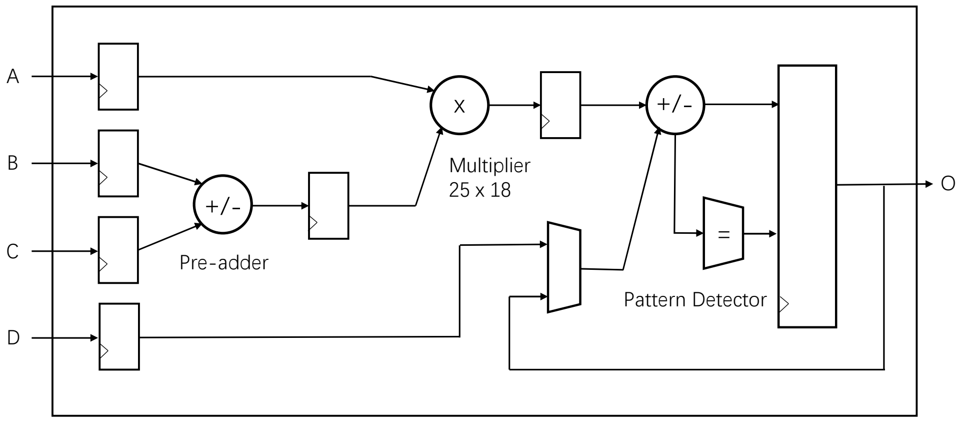


Figure 2.3: The example of DSP block structure

to increase the performance of FPGA. In addition, the BRAM can be used to realize a hardware FIFO, which is a very useful structure in hardware design.

2.2 Traditional FPGA Developing Methodology

In this section, I first introduce the overview of FPGA design flow. Then, I present two typical developing methods of FPGA design, i.e, traditional register-transfer level (RTL) based design, and high-level language based design. I also compared these two design methods by their features.

2.2.1 FPGA Design Flow

The design flow of FPGA is very similar to the IC hardware design [21,22], since FPGA can be treated as a special IC chip that can change the design after fabrication. Figure 2.4 shows the overview of FPGA design flow.

The design flow starts with the design entry process. Initially, when the size of hardware circuit is not large, the design entry can be described very specifically by using the gates and how they are connected by wires, i.e., schematic entry. However, with the development of FPGA technology, more hardware resources are integrated on an FPGA, the schematic design entry become hard to maintain and the developing productivity is extreme low.

As the design complexity continues to grow, the design entry of FPGA starts to use more abstraction level languages to describe the design structure. Hardware description languages (HDLs), e.g., VHDL or Verilog based RTL design is a significant advance in terms of abstraction. Compared with the description of gates and wires, the RTL design allow developers to use HDLs for declaring the digital logic behaviours and operations based on registers. The RTL-based design method is also known as the traditional design method of FPGAs, I will discuss the detail features in the following section.

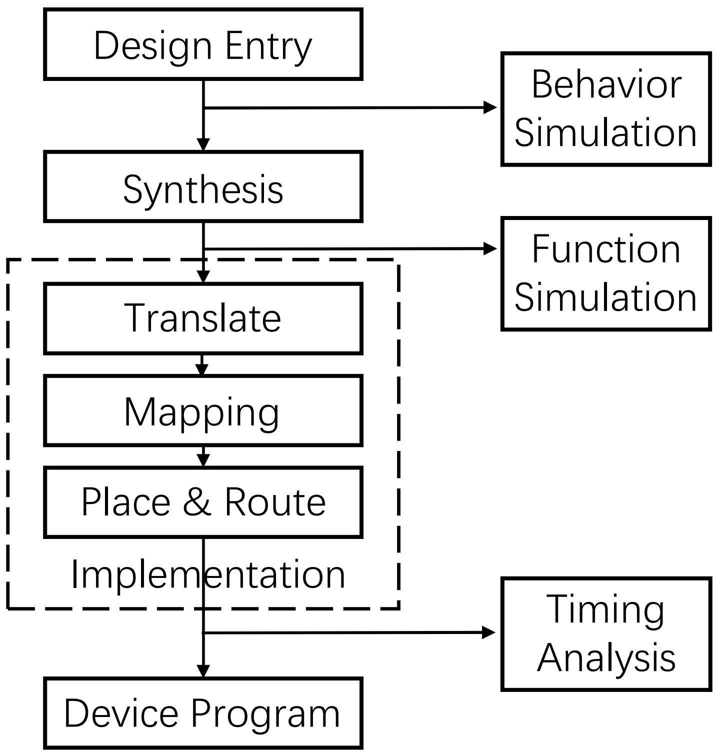


Figure 2.4: The design flow of FPGA

Before the developers actually program the FPGA chip with the binary files, i.e., bitstream. They need to use the corresponding electronic design automation (EDA) tools such as Vivado or Quartus to help them generate the target bitstream. The EDA tools mainly contain the following 3 parts: synthesis, implementation, and device programming.

After finish the design entry process, the synthesis process actually transfer the HDL-based RTL design into gate level netlist. Not all HDL code can be synthesised to gate level, such as, duel clock triggering process, or recursive functions. The synthesis tool can detect these descriptions and check the code errors.

Next, the gate level design will go through the implementation process which consist of 3 sequence steps.

- First, the translation step binds the gate level netlists and their time requirements together to a design file.
- Then, the mapping step cut the netlists into small logic elements and mapping the logic elements with FPGA hardware resources, e.g., CLBs, DSPs, or BRAMs according to the design file. For example, Figure 2.2 shows how to map a "AND" logic to a CLB block.
- Finally, the place and routing (PAR) step actually locates the mapped hardware resources inside the FPGA (place) and connecting these blocks with wires and

routers (routing). However, the PAR may generates congestion and wires can be detoured. As a result, the output solution of PAR does not always meet the area or time requirements of target design. Generally, the PAR can generates multiple solutions and choose the best one as the final output. If the final solution still fail to meet the requirement, the developers need to optimize the design entry file and repeat above processes all over again.

After the implementation, the target design file is transformed to a bitstream format that can be downloaded to a specific FPGA. This process is done by the device programming. The total FPGA design flow end here.

2.2.2 RTL-based Design Method

The traditional development method begins with RTL design [23]. Developers use HDL language to describe the behavior of logical functions between registers. Before entering the synthesis phase, they need to perform RTL simulation to ensure that their HDL code works as expected.

However, the logic of HDL language is not easy to grasp. It usually requires deep hardware design knowledge. For example, it is not recommended to use a latch in RTL design, but developers can misuse an incomplete "if else" statement to generate the latch structure which is an asynchronous storage element and may cause time error. In another case, the RTL design needs to control the clock signal to perform synchronization or implement the control logic of the data path. Only when the clock signal reaches the register with exact predefined clock cycles, this signal can be treated as receiving correctly. When the clock signal is faster or slower even by 1 clock cycle, the behavior of the RTL design may be completely different. As a result, generating correct RTL simulation results may require a lot of time to verify the HDL code.

After the developer has verified the HDL code, the EDA tool can be used to synthesize and implement the design. However, the verified design may also need to change since the PAR fail to meet the time and area requirements as I state in previous section. Then, the developers must optimize the HDL code and restart the verification and EDA process. Therefore, the developers are usually required to coding a large amount of HDL code for a target design and pay special attention to the details of verification and implementation. The result is that developers spend a lot of time on RTL code instead of proposing new solutions and evolving new algorithms.

With the continues evolution of FPGA technology, the productivity and usability of RTL-based development methods cannot overcome future challenges, e.g., time-to-market. In order to remain competitive, development methods based on high-level languages began to gain popular. I will introduce the HLS-based design method in the next section.

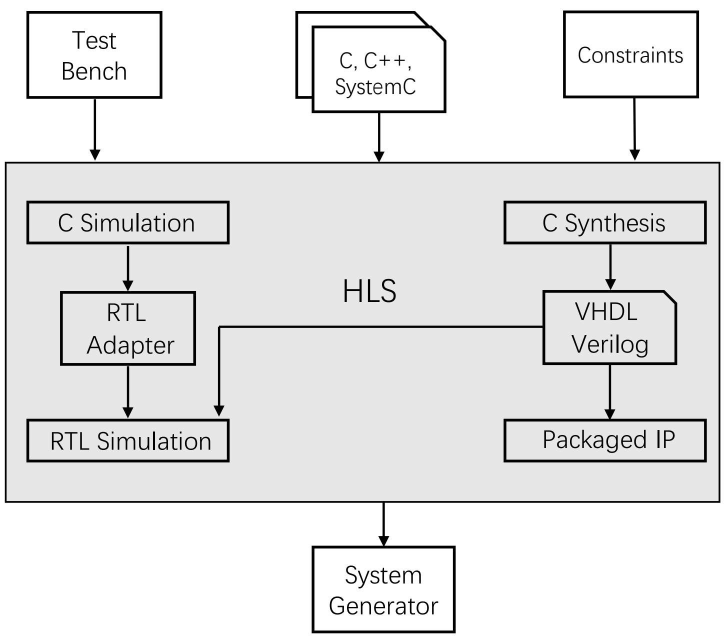


Figure 2.5: The HLS design flow

2.3 High-Level Synthesis FPGA Developing Methodology

Similar as the RTL-based design method replacing schematic method, the high-level synthesis developing method is a further step to improve the design description level from RTL to algorithmic [24]. Compared with the RTL description, the HLS developing method allow developers to implement the arithmetic-level behavior by directly using high-level languages such as C or C++. Since the developers work at a higher abstraction level and do not need to care about the detailed design of the data path structure and its corresponding control logic, the development time is significantly reduced. Through the HLS design flow, developers can automatically convert arithmetic-level designs to RTL-based designs, such as transforming C code to HDL code. Once the RTL-based design is available, developers can use the standard FPGA design flow to generate bitstream files and program the target FPGA as shown in Figure 2.4.

2.3.1 HLS Design Flow

A typical HLS design flow is shown in Figure 2.5 [25]. According to the input high-level design files and constraints, the HLS compiler can use HDL code to generate RTL designs. Basically, the HLS compiler automatically performs the following tasks instead of explicitly performing as in RTL design. The HLS compiler can generate a design interface based on the data type of the parameters of the defined function. Inside the function, HLS analyzes data structures, calculation operations and control statements. Based on this, HLS can generate the corresponding data path and control logic. By using the constraint file, HLS can also pipeline the design and insert registers into the critical path to optimize frequency. As a conclusion, the final target of HLS design flow is to automatically optimize the high-level design based on the constraints and generate the corresponding RTL design. The HLS process mainly uses the following 3 phases to reach the design target.

Scheduling and Binding

The main task of the scheduling is to solve the problem of how to assign the computing operations of a high-level function to the correct clock cycle stages. Based on the constraints, the scheduling system can assign multiple computing operations inside one clock cycle. Meanwhile, if the target clock period is too short, the scheduling system also can assign the operations across multiple clock cycles. The detail scheduling strategies can be seen in paper [26]. After the scheduling step, the binding system decides which type of hardware resources, e.g, CLB or DSP, to execute the scheduled computing operation. For some highly optimized design, developers also can use directives to directly bind the proper hardware resource to the specific operations.

I use the following example to explain how the scheduling and binding work. A high-level function is shown in List 2.1.

```
//a, b, c are constant numbers.
int foo(int input)
{
  int result;
  result = a * input + b + c;
  return result;
}
```

Listing 2.1: An HLS code example

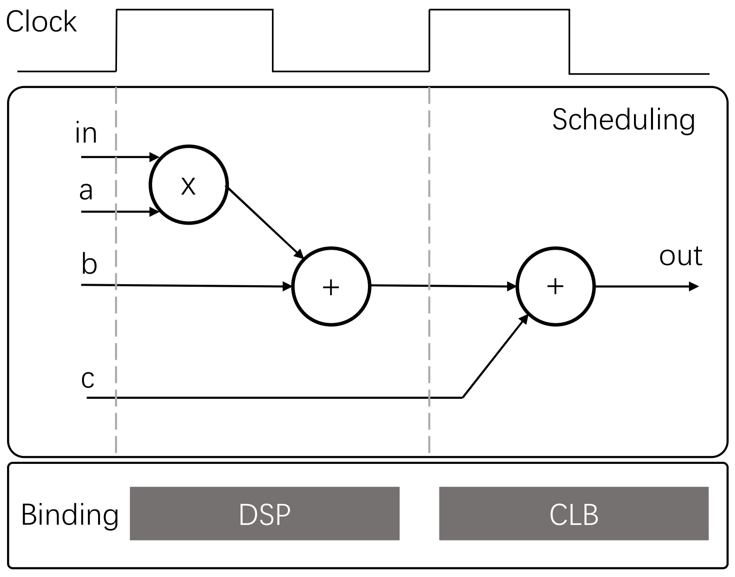


Figure 2.6: The scheduling and binding result

The result of scheduling and binding is shown in Figure 2.6. Inside the example function, there exists 3 computing operations. The scheduling system assigns the multiplication and addition in the first clock cycle. The other addition is scheduled to the second clock cycle. After the scheduling phase, the bind system binds the multiplication and addition to the DSP block, since DSP block can efficiently execute the multiply-add (MADD) operation compared to CLB. The next addition is implemented with the CLB.

Control Logic Extraction

As I state above, the HLS can analyze the conditional statements of high-level language and automatically generate the control logic. I use a code example to show the idea of control logic extraction.

```c
//a, b, c are constant numbers.
#define N 3
int foo(int input[N], int out[N])
{
  int t;
  for(int i = 0; i < N; i++)
  {
   t = input[i] * a + b + c;
   out[i] = t;
  }
}
```

Listing 2.2: An code example of a loop statement

For the target foo function, there are two arguments are declared as the integer array type. The array in the C language can be implemented by using BRAM resources. The HLS can automatically generate the control signals, e.g., read or write enable to access the I/O interface of the BRAMs.

Inside the function, a loop conditional statement is used to control the program to repeat N times of the 3 computing operations and write the result value to the output array. The HLS can automatically create the finite state machine (FSM) to implement the loop statement. This FSM is shown in Figure 2.7.

Since the result of adding operation of $b+c$ can be reused inside the loop statement. The HLS can schedule this addition out of the loop and be a separate state of FSM, i.e., $S0$. Then, in the next state $S1$, it keeps the result of the addition $b + c$ in the register (FF) and starts to enter the loop structure. $S2$ is used to wait the return value from the BRAM, since the memory access of BRAM usually need to cost 2 clock cycles. At last, in the state $S3$, it calculate the result of output by binding the MADD operation to the DSP. In addition, it also checks the loop boundary of counter i. If i exceeds the value N, $S3$ goes back to the start state $S0$, otherwise, it goes back to the state $S1$. As a conclusion, in case of $N = 3$, the order of the FSM is:

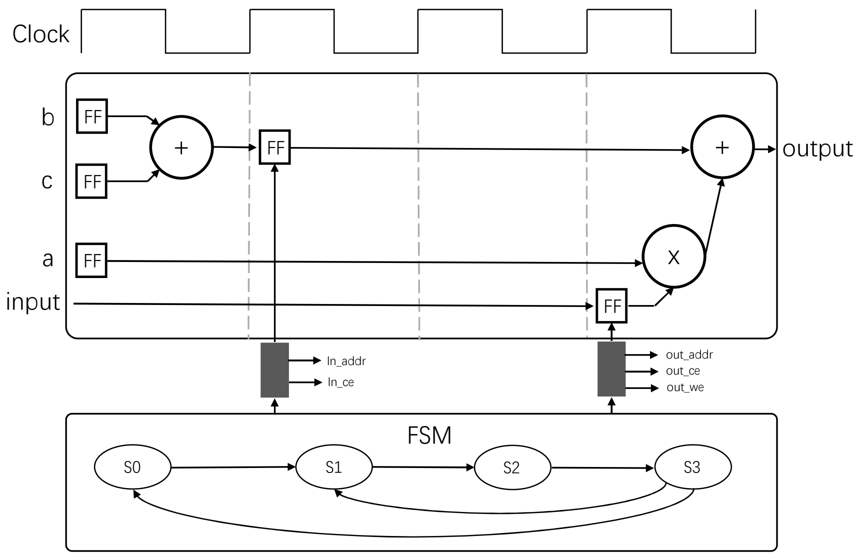


Figure 2.7: The FSM to control the loop statement

$S0 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S0.$

2.3.2 HLS Developing Tools

The development of HLS tools can be traced back to around 1990s. At this stage, these HLS tools try to use more abstract design which is often described by the custom HDL to generate the RTL design. For example, the academic studies include, e.g., MIMOLA [27], ADAM [28], HAL [29]. And the related commercial tools are Behavioral Compiler from Synopsys [30], Visual Achitect for Cadence [31], and Monet with Mentor [32].

However, These HLS tools cannot be considered successful. These early HLS tools are only used for design prototyping and academic exploration. This is mainly caused by the following reasons:

- Instead of using the conventional high-level languages, e.g, C or C++, these HLS tools choose to modify the HDL languages, e.g., removing the explicitly clock signal, to support the high level abstraction. Although C or C++ are very popular languages, they are not considered suitable for hardware design, due to their complex memory management, e,g., pointers, lack of parallel scheduling, and explicitly synchronization system. Compared with C, the custom HDL languages require users to spend a lot of times to manage them. Moreover, these tool-dependent custom HDLs are very hard to port them to other platforms.
- Early HLS tools pay much attention on the data path generation. They still need users to solve the system integration problem by themselves. For example, users often need to specify the connecting interfaces from target synthesised function to other system modules, which hurts the design reusability.

- Due to lack of verification techniques support, the quality of results (QoR) of these HLS tools are very awful. The generated RTL design of HLS has few feed backs information to the high-level user design. For example, these HLS focus on optimizing the latency result, rather than considering the corresponding area or power costs, making the target design often can not be correctly implemented with the limitation of power or area budgets.

With the decades of efforts from both academia and industry, state-of-art HLS tools can solve these shortcomings to a certain extent and effectively generate the high QoR RTL design directly from the high-level languages, e.g., C or C++. I will introduce some typical HLS tools in the following.

On the industry side, both Xilinx and Intel the two big vendors of FPGA have proposed their HLS compilers for implementing C-based design on their FPGA products. Vivado HLS [13], the HLS tool of Xilinx, is derived from the AutoESL which is founded by UCLA vast lab. Vivado HLS can package the the high-level abstraction of algorithmic description to the IP core without manually creating RTL. As a commercial tool from Xilinx, it can efficiently utilize the on-chip DSPs, memory resources, and predefined libraries. It also can verify the C-based design by using C level test bench.

The most successful HLS tool of Intel is the OpenCL SDK [33]. Compared with Xilinx, Intel is not only the vendor of FPGAs (acquiring from Altera) but also the leader of CPUs. Intel OpenCL SDK exploits the features of OpenCL, a open standard defined by Khronos Group for heterogeneous computing devices. The goal is to provide an uniform developing environment for the heterogeneous platforms, e.g., multi-core CPUs, FPGAs, or even GPUs. By using the board support package (BSP), many third party of FPGA boards can also use the OpenCL DSK, and generate the RTL design by using the OpenCL C language.

For the academic side, there also exist many HLS tools. Compared with commercial tools, these academic tools are usually open source and do not bind with specific FPGA vendors. LegUp [15] is an HLS tool invented by J. Anderson from the University of Toronto. Similar like other HLS tools, LegUp can generate the RTL result according to the high-level languages. In addition, it also supports the heterogeneous computing systems as the Intel OpenCL SDK, and it can use various CPU types such as ARM, or MIPS architecture. By using another open source compiler and tool-chain LLVM [35], LegUp also can work with the multi-threads API, e.g., OpenMP and the Pthread, allowing users to directly improve the performance of FPGA by utilizing the software-based parallel processing techniques.

Bambu [34], a free HLS framework for the complex applications, is developed with the Politecnico di Milano. It supports many C features, which is rare compared to other HLS tools, such as pointer arithmetic, dynamic resolution of memory accesses, and custom C struct data types. Bambu is implemented with C++ language and the different stages of HLS process are specified in their corresponding C++ classes, making Bambu a lightweight modular tool. Although, Bambu does not provide the corresponding verification tools, e.g., test bench simulators, the generated HDL files are compatible with other commercial simulators.

Chapter 3

Stencil Computation and Related Work

In this chapter, I introduce the background knowledge of stencil computations and demonstrate two typical parallelization strategies, e.g., spatial-based and temporal-based methods to improve the performance of stencil kernels in the first section. Then, in the next section, I present and discuss the previous studies of how to achieve high performance for stencil computations on FPGAs by using the HLS developing methods. At last, through the discussion of these related works, I found that there are still some challenges that have not been resolved. I briefly show how I gonna solve these challenges with the proposed solutions in the following chapters.

3.1 Background

In this section, I first use an detailed example to explain the definition of stencil computations. Then, I explain how to exploit the parallelism of stencil kernels since the performance is highly related to the level of parallelism.

3.1.1 Stencil Computation

Stencil computations are a class of kernels using the fixed algorithm pattern (stencil) to update the target data elements on the one or multi-dimensional grids [36]. Stencil computations have been widely used in various HPC applications, such as fluid dynamic and electromagnetic simulations [37,38], solving mathematical equations by discretizing the time iterations [39], and computer vision [40].

A detailed example is shown in Figure 3.1. The target grid is stored in an 2D array where the length of column is N and the length of row is M. The target data is represented by using the black block with the index (y, x). To calculate the data in (y, x), the data from four neighbor blocks, i.e., $(y+1, x), (y-1, x), (y, x+1), (y, x-1)$

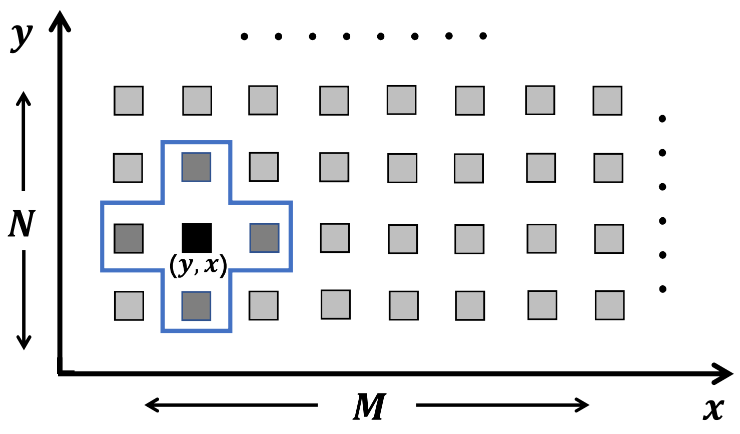


Figure 3.1: Example of a four point stencil kernel on a $(M \times N)$ 2D space

are required. The updated target data is equal to:

$$I_{t+1}(y,x) = 0.25 * (I_t(y+1,x) + I_t(y-1,x) + I_t(y,x+1) + I_t(y,x-1)) \tag{3.1}$$

where I_t represent the iteration time-step. After finishing all blocks in the target array, the stencil kernel can move to the next iteration step. I can easily use several lines of high-level C code to implement this kernel in Listing 3.1.

```
1 for(time_iter=0; time_iter<I; time_iter++){
2  for(y=0; y<N; y++)
3  for(x=0; x<M; x++){
4  out[y][x] = (in[y][x+1]+in[y][x-1]
5     +in[y+1][x]+in[y-1][x])*0.25f;
6  }
7  swap(out, in);
8 }
```

Listing 3.1: High-level code for Laplace equation kernel

As shown in Listing 3.1, the target grid size is $M \times N$. The data in the target grid are stored in the array ”in” and “out”. The function ”swap” is used to transfer data from the array ”out” to ”in”. The iteration time-step is defined by the value I. The 3 loops (line 1, 2, 3) are the control statements of the C code. The first one traverses the time step from 0 to the I. The second and third loops traverse the 2D grid space. The target stencil operation is executed in line 4 and 5. These nested loops indicate the stencil kernel is highly parallelizable. In the following section 3.1.2, I will demonstrate the parallel computing strategies of stencil kernels.

3.1.2 Parallel Computing Strategies for Stencil Computation

Regardless of the type of computing devices, e.g., CPUs, GPUs, or FPGAs, the key to achieve enough high performance on stencil computations is to run the stencil operations in parallel as much as possible, i.e., utilizing the parallelism of stencil kernels. For example, in the Listing 3.1, the maximum parallelism value can be calculated as $I \times (N \times M)$. According to the loop executing dimensions, the total parallelism can be divided into two types, one is based on space dimension, i.e., spatial parallelism and the other is based on time domain, i.e., temporal parallelism. These two kinds of parallelism are the foundation to build parallel stencil computing structures. The relationship between the performance and these parallelism is described by:

$$P_{stencil} \propto (Spatial\ Parallelism \times Temporal\ Parallelism) \tag{3.2}$$

where the $P_{stencil}$ is the stencil computing system performance.

Spatial-based Strategy

In Listing 3.1, the loop statements in line 2 and line 3 traverse the 2D array space by using the row order. From the high-level abstraction, the parallel computing strategy can unroll the two loops, such as along the x-axis, y-axis, or even both axes, meaning that the computing system wants to execute the multiple stencil operations simultaneously within the same time iteration. The number of parallel executing stencil operations represents the spatial parallelism. And this kind of parallel computing method is often referred to as spatial-based parallelization strategy.

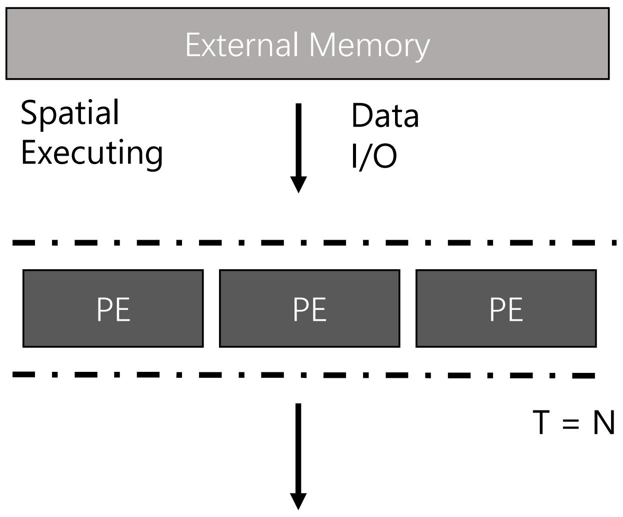


Figure 3.2: Spatial-based Parallelization Strategy

Figure 3.2 introduces an example for implementing the parallel computing structure with the spatial-based strategy. Since the array used to store the stencil data usually is too large to be located in the on-chip memory resource, e.g., cache, or BRAMs, it usually need to be stored in the external memory such as DRAM banks. The processing elements (PEs) are responsible for executing the stencil operations. Adopting spatial-based strategy means to schedule all PEs inside the space domain, i.e., at one iteration time-step.

To achieve high performance with the spatial-based strategy, there are two constraints the users need to consider. Although increasing the number of PEs need to consume more hardware resources on the target computing device, the PEs also requires enough memory bandwidth to access the external memory with the I/O interface. Moreover, compared with computing resources consumption, the stencil computations are often dominated by the memory accession. For example, in List 3.1, assume the users do not employ any optimization methods, it need to access 4 data from external memory to calculate one stencil operation.

For above reasons, only adopting spatial-based parallelization strategy on the devices with low external bandwidth often can not reach the peak performance of these devices. For instance, in [41], a stencil computation is implemented on an AMD CPU, which only utilizes 10 percentage of the CPU peak performance. Even for the GPUs whose external bandwidth are often 10 times higher than CPUs and FPGAs, using spatial-based strategies also can not reach their peak performance. Indeed, through some optimization methods, the memory bandwidth requirement for the stencil computations can be reduced. I will discuss these optimization methods in the next related studies section.

Temporal-based Strategy

Similar like the spatial-based strategy unrolling the loops in line 2 and 3, the temporal-based parallelization strategy choose to unrolling the loop structure in line 1 which traverses the whole time dimension. Unrolling this loop in the target design means that FPGA can execute stencil operations in parallel at different time iterations, especially in sequential order. The number of parallel computing stencil operations represents the temporal parallelism. This kind of parallel computing method is often referred to as temporal-based strategy.

Figure 3.3 explains the conception of parallel computing structure with the temporal-based strategy. 3 PEs are running in the iteration time-step $N-1, N, N+1$. Compared to spatial-based strategy, only 1 PE needs to read or write data to the external memory. The other PEs access data from their previous time-step PEs and pass the outputs to the next time-step PEs. As a result, to increase stencil computation performance, i.e, the number of PEs by using the temporal-based strategy do not need additional memory bandwidth support. The maximum value of temporal parallelism comes from the device hardware resources.

However, the main challenge of using temporal-based strategy comes from the data dependency issue. For stencil computations, data dependency means the current iteration's data is dependent on the previous iteration time-step. For example, in Figure 3.3, to calculate the result of target stencil operation on the black block N, it needs

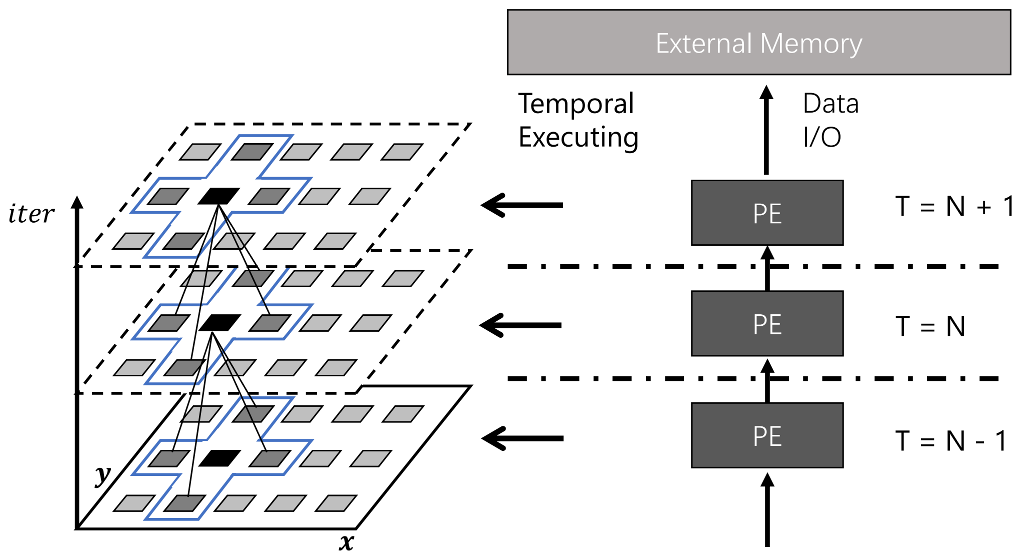


Figure 3.3: Temporal-based Parallelization Strategy

data from 4 deep gray blocks from the previous time-step $N-1$. Similarly, to get the result of black block in $N+1$, the 4 deep gray blocks in iteration N is required. Assume the users want to insure that the PES in time-step $N-1, N, N+1$ run fluently without stalls, a well-designed cache system and the connection interface between the PEs need to be considered. Compared to fixed architecture, e.g., CPUs, GPUs, FPGAs have large on-chip memory resources. Moreover, these memory resources can be customize according to the specific applications. Many previous studies have shown some solutions. I will discuss them in the following section.

3.2 Related Studies

3.2.1 General Stencil Computation Optimizations

As I described in the section 3.1.2, compared with utilizing the temporal-based parallelization, using the the spatial-based strategy is a more intuitive method, since there is no data dependency issues in space dimension. To achieve the high performance, the bottleneck mainly comes from the external memory bandwidth. The on-chip memory resource of computing devices can not store the target stencil array. Therefore, users need to store the stencil array in external memory and access data by the I/O interface. In addition, the arithmetic intensity of stencil computation is usually very low, i.e., one stencil operation needs multiple data accession. And the memory access pattern is not sequential and hard to coalesce to use the wide I/O bus width.

To remove the performance bottleneck from memory bandwidth, many previous studies [36, 42, 43]choose to specifically cache one part of the target stencil array to the on-chip memory structure, e.g., cache system. Inside the space dimension, there

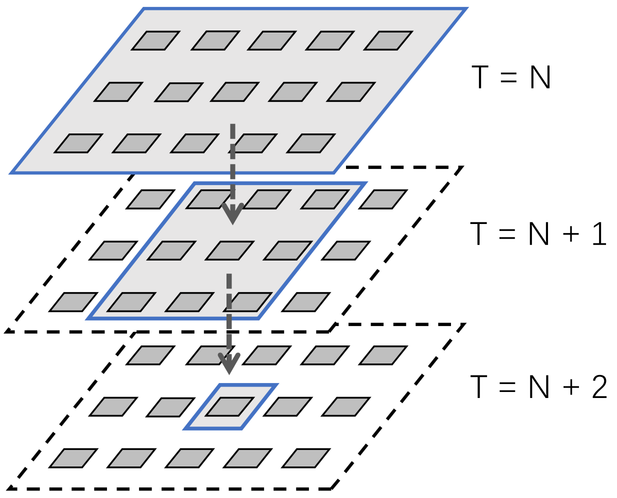


Figure 3.4: The temporal blocking example

exist strong data reuse opportunities for stencil computations. For instance, in the time-step N, performing stencil operation on index (y, x) needs to read 4 data, i.e., $(y+1, x), (y-1, x), (y, x+1), (y, x-1)$. On the contrary, the data on index (y, x) can also be reused in performing 4 stencil operations on the index $(y, x+1)$ as left side neighbour, $(y, x-1)$ as the right side neighbour, $(y-1, x)$ as the up side neighbour, and $(y+1, x)$ as the down side neighbour. Assume the cache system is large enough to hold all stencil data in one space dimension, then, the arithmetic intensity of stencil computation can be optimized to 1 data accession for 1 stencil operation. Obviously, the on-chip cache is a limited memory resource, these previous studies use various tiling methods to solve this problem. e.g., diamond, split, or wave-front tiling.

However, buffering data from one space dimension only takes advantage of the data locality in space area, e.g., 1 data can be mostly reused 4 times in Listing 3.1, which hurts the design scalability. For some stencil computations, their performances are still limited by the external memory bandwidth. To solve this problem, [44–46] propose the temporal blocking solutions to not only benefit data reuse-ability in the space dimension but also in the time dimension.

Similar as the previous work, temporal blocking also buffer one part of stencil array in the cache to perform stencil operations on these data. However, after finishing the computation of one space dimension, they do not immediately write the results back to the external memory as in the previous solutions. The temporal blocking still keeps the calculated results in the cache, and start performing stencil operations on these data to generate results for the next time-step. The temporal blocking method can repeat multiple times, it only writes the last results to the external memory. Through the temporal blocking, the stencil data can be reused across multiple time steps. As a result, the external memory bandwidth requirement for stencil computations can be reduced significantly.

Figure 3.4 shows an example of temporal blocking technique. One thing to be noticed here is the valid temporal blocking outputs is smaller than the inputs. For instance, there exist 15 valid outputs in time-step N. In the time-step $N + 2$, only one valid output remains. To avoid frequently access the external memory, the boundary stencil cells can not be correctly updated with the increased time-step because they do not have enough valid data to perform the stencil operations. This situation often causes two problems. One is that to use temporal blocking across many time-steps, the on-chip memory resources need to store large area of overlapped data, resulting in a lot of redundant calculations. The other one is that for architecture like GPUs, the irregularly outputs in every time-step can cause thread divergence-i.e., every thread in a warp needs to execute the same instruction within the same period.

3.2.2 Optimizations with FPGAs

Compared to computing devices, e.g., GPUs, the external bandwidth of FPGAs is usually very low. Although data reuse opportunities of stencil computations both in space and time dimension can be exploited to reduce the memory accession from external memory, applying spatial-based parallelization strategy on FPGAs often can not reach the peak performance of FPGAs.

On the other hand, exploiting temporal-based parallelization strategy for implementing stencil computations on FPGAs is very convenient. Building custom pipeline structure is one of the significant advantages of the FPGA architecture. Through customizing a pipeline system, developers can overcome the data dependency issue in the temporal-based strategy, which makes parallel computing stencil operations in different time-steps possible. As a result, the stencil computation performance can scale-up along the temporal parallelism with constant external memory bandwidth. Many significant studies [47–50] have proposed optimizations based on the temporal-based strategy, especially in the area of memory partitioning algorithm, data reuse cyclic buffer, overlapped tiling combined temporal-based method, and stencil computation design automation.

Studies [51–53] propose stencil architecture on FPGAs with the temporal blocking optimization method. They choose to use coarse-grained parallel computing structure like the thread parallelism in multi-core CPUs or GPUs. Compared with thread-based temporal blocking implementations on fixed architecture, the thread divergence can be avoided by utilizing the flexible architecture of FPGAs. Since there exist more valid stencil operations in early time-steps than in the later time-steps (as I stated in Figure 3.4), The hardware resources of FPGA can be allocated according to the different time-step requirements. For instance in Figure 3.4, 15 times computation units can be used in time-step N compared to time-step $N + 2$. However, there still have large overlapped area and redundant computations in these studies. Moreover, they do not fully utilize the fine-grained pipeline system on FPGAs, which lose the further optimization opportunities.

Although the temporal-based strategy mainly exploits the data locality in the time dimension, the data reusability in the space dimension is also important. As I discussed at the beginning of this section, buffering data in the cache structure can solve this problem. Compared to CPUs or GPUs with fixed cache structures, FPGAs usually need to use the on-chip memory resources, e.g., BRAMs to build a custom buffer structure according to the target applications. The application-specific buffer has many advantages. Unlike the general purposed cache system, the custom buffer structure can decide which data should be located in the buffer and adopt a unique buffer update program. This is can avoid problems such as cache misses or false sharing, i.e., when multiple computing units update different data elements in the same cache line, they will interfere with each other. However, the efficiency of custom buffer structure is highly dependent on the developer's skills. Inappropriate buffer design may cause serious performance degradation.

To build an efficient custom buffer structure, paper [47] first proposes a memory partitioning method to assign the buffed stencil array into different BRAMs. Since the stencil operation usually needs to parallel access multiple data, partitioning these data into different BRAMs allows computing unit to read or write these data in one single clock cycle, which increases the stencil computation performance. Later, the other research [48] introduces a cyclic custom buffer structure to optimize the memory partitioning method. Instead of buffering large amount of stencil data to exploit the data locality in space dimension, this method choose to only buffer data with minimum data reuse distance. For example, in Figure 3.1, the data in index $(y-1, x)$ is first accessed as the up side neighbour for performing stencil operation in $(y-2, x)$. After reuse the data as left side and right side neighbour, the last accession of this data is to use it as the down side neighbour to perform stencil execution on index (y, x). Then, the minimum date reuse distance is between the index $(y-2, x)$ and (y, x), i.e., equal to $2M+1$ where M is the row length.

An HLS-based stencil computation design flow is proposed in paper [51]. Their design architecture is developed with Vivado HLS and Synphony C. They utilize the temporal-based parallelization strategy to increase the performance of target stencil applications. Their cone-based custom buffer design portions a frame of the stencil array. Compared to the buffer design in [48], their cone-based design cost more on-chip memory resources to exploit the data reusability. In addition, their design do not discuss the possibilities to scale the stencil performance with spatial-based strategy.

The studies in [49,50] employ HLS developing method to implement the high performance stencil computation architecture on FPGAs. They choose to use Intel OpenCL SDK as the target HLS tool. The cyclic custom buffer method is also adopted in their implementations. With the HLS tool supports, they abstract the cyclic buffer behaviour as a shift register structure. Then, they connect the shift register buffer to realize the temporal-based parallelization strategy. In addition, the [49] propose a overlapped tiling combined temporal blocking method. As I discussed above, this method can use redundant computation as a trade-off to reduce the data reuse distance, thereby saving the on-chip memory resources. However, their shift register behaviour abstractions only work well with the Intel OpenCL compiler. I will show the detailed reasons in the following section.

In research [54, 55], they propose the similar stencil computation architecture with previous [49, 50]. Specifically, they discuss the detail techniques of how to connect the custom buffer to form a data-flow or streaming structure. Furthermore, [54] introduce a open source HLS optimization library for better employing the common functions on FPGA. [55] provides a design automation tool of stencil computations to further increase the design productivity. However, due to the external bandwidth limitations of tradition FPGAs, they can not fully explore the design space and miss the potential computing results reuse optimizing opportunities for some stencil computation applications. I will also discuss this problem in the next section.

3.3 Challenges

Through the summary of the above-mentioned studies, I found that there still exist 3 major challenges which have not been perfect solved or not solved. In this section, with the help of a specific example, I will explain and analysis these 3 challenges, i.e., shift-register custom buffer behaviour (section 3.3.1), sub-optimal hardware resource consumption in spatial-based strategy (section 3.3.2), and the computing result reuse opportunities (section 3.3.3).

The previous studies can be concluded using the pseudo code in the Algorithm 1. The stencil computation of Listing 3.1 is implemented in the loop (*while*) structure from line 5 to 32. This structure is mainly composed of 2 parts. One part is used to build the on-chip memory system, i.e., the cyclic custom buffer. The behaviour of the cyclic buffer is abstracted as the shift register described from lines 7 to 13. The other part is the computing unit (from lines 14 to 22), which is used to execute the stencil operation by accessing the data in the shift register.

There also exist 3 pragmas (*pragma unroll*) in the loop structure pseudo code. These pragmas are used to implement the temporal-based parallelization strategy. In default, the loop structure in the high-level code is executed in sequence. The HLS tools can automatically generate the corresponding data path and control logic for implementing each loop iteration in order. Assume developers want to increase the performance of loop structures, the *pragma unroll* can guide the HLS tools to automatically generate the logic for parallel executing the loop iterations. The first two unroll pragmas in line 7 and line 9 are used to duplicating the shift register structure and make sure the shift register can finish shift operation in one clock cycle. The pragma in line 15 is to duplicate the computing unit.

Algorithm 1 Pseudo Code for Implementing the Stencil Computations in [50]

```
procedure STENCIL(din, dout)
    shiftreg[$d$][$size$];
    result[$size$];
    Loop iterations = $x$;
    while $count \neq x$ do
        for $i = size - 1 \rightarrow i = 1$ do
            # pragma unroll
            for $j = 0 \rightarrow j = d$ do
                # pragma unroll
                shiftreg[$j$][$i$] = shiftreg[$j$][$i - 1$];
            end for
        end for
        shiftreg[0][0] = din[$count$];
        for $j = 0 \rightarrow j = (d - 1)$ do
            # pragma unroll
            \\boundary conditions
            $U$ = (condition 1) ? shiftreg[$j$][0]:0;
            $R$ = (condition 1) ? shiftreg[$j$][$M - 1$]:0;
            ...

            \\computation
            result[$j$] = $0.25f \times (U + R + D + L)$;
            if $count > (j + 1) \times latency$ then
                if $j == (d - 1)$ then
                    dout[$count$] = result[$j$];
                else
                    shiftreg[$j + 1$][0] = result[$j$];
                end if
            end if
        end for
        $count++$;
    end while
end procedure
```

3.3.1 Shift-Register based Custom Buffer

The first challenge comes from the high-level shift register-based custom buffer design. Figure 3.5 shows a shift register buffer design for the stencil pseudo code in Algorithm 1. The stencil cells covered by the blue color are stored inside the shift register. The index of each stencil cell in the buffer is based on the lexicographic order which is linearized the 2D index to 1D from right to left and up to bottom.

The behaviour of the shift register is describing as following. For every clock cycle, the shift register buffer shifts in the fresh stencil data from the input interface to the

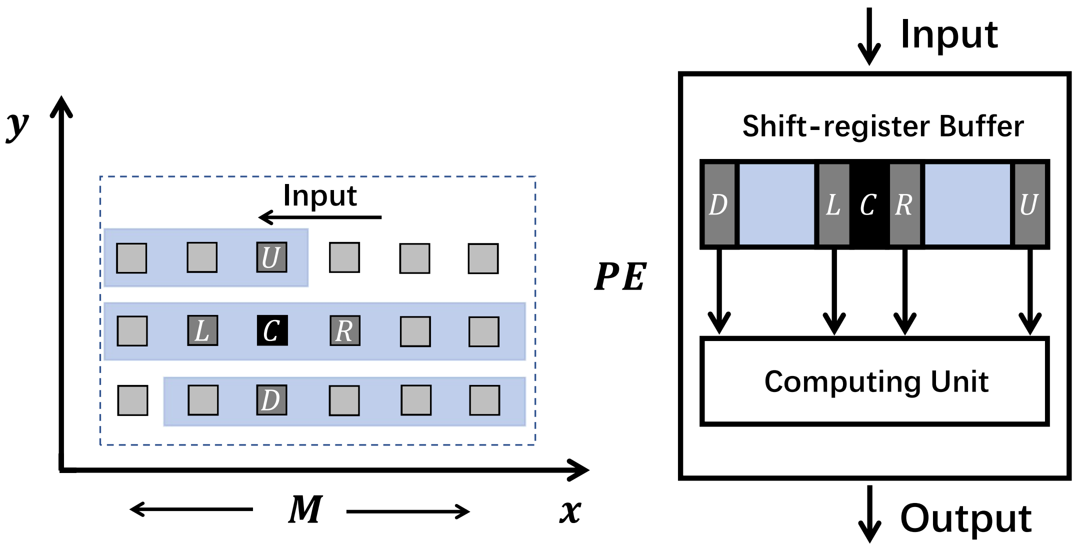


Figure 3.5: The shift register-based custom buffer design

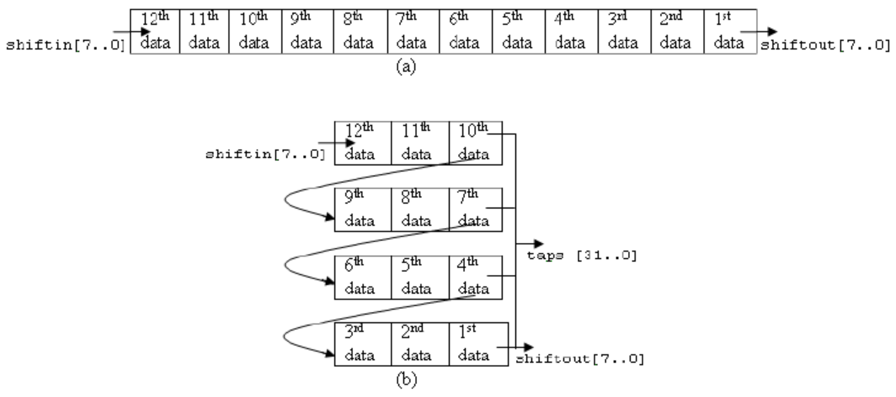


Figure 3.6: The Intel Shift Register IP Core [56]

head location ($index = 0$). Meanwhile, the last used stencil data ($index = 2M$) is shifted out from the shift register. To perform the stencil operation, the computing unit needs to access 4 stencil cells, i.e., U, R, L, D. These data are located in the index 0, $M-1$, $M+1$, $2M$ respectively. However, this kind of behaviour only can achieve the maximum performance under the Intel-based HLS tools. This is mainly due to the following reason.

For the commercial HLS environments, they usually have the ability to exploit their high efficient predefined shift register IP cores. Their HLS tools usually can recognize the high-level shift register behaviour code pattern and synthesis this part of code by directly using the corresponding IP cores. However, the IP cores from different developing environments may have inconsistent standard. For example, Figure 3.6 shows the IP core design from Intel environment. The Intel shift register IP core allows developers to access the internal data of the shift register in parallel, i.e., the 4 data $1_{th}, 4_{th}, 7_{th}, 10_{th}$ can be accessed through the taps. Compared with Intel, Figure 3.7

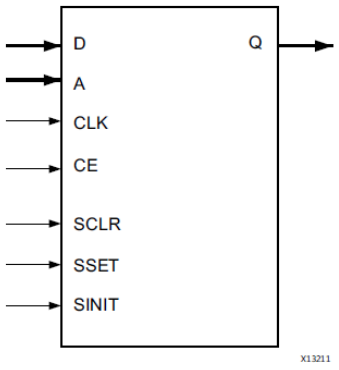


Figure 3.7: The Xilinx Shift Register IP Core [57]

shows the Xilinx shift register IP core interface. The developers only can shift data in via the "D" port and shift data out through the "Q" port. The internal data can not be accessed. As a result, the shift register-based custom buffer design is strongly dependent on the Intel HLS tool.

For overcoming this limitation, I introduce a custom buffer design that explicitly described by using the FIFO and register elements. Unlike the shift register IP core, the behaviour of FIFO and register are quite common in HLS environments (e.g., push and pop). For example, both Intel and Xilinx have the similar FIFO IP behaviours (hls::stream for Xilinx, or ihc::stream for Intel). In Chapter 4, I show this work in detail with the specific LGCA application.

3.3.2 Optimal Custom Buffer Design for Spatial-based Parallelization

The second challenge is that the custom buffer design in previous work can only support one computing unit running at full speed, i.e., executing 1 stencil operation. To perform multiple stencil operations, the pseudo code in Algorithm 1 exploit the *pragma unroll* to duplicated the shift register-based buffer along with the computing unit, as shown in Figure 3.8.

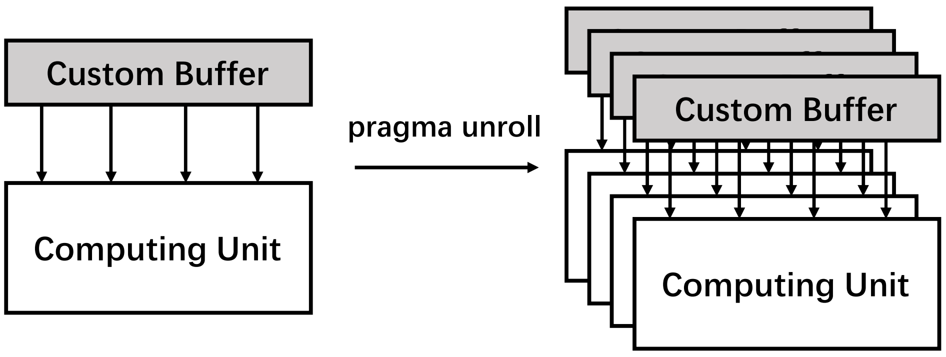


Figure 3.8: The overview of unrolled custom buffer design

However, the efficiency of this kind of duplication is very low, especially in the aspect of the resource consumption when the multiple stencil operations are executed in the space dimension, i.e., at the same iteration. The computing unit is responsible for performing the stencil operation. Thus, duplicated computing units can perform multiple stencil operations in parallel. Meanwhile, the custom buffer mainly have two tasks. One is to store the stencil data for exploiting data locality. The other one is to provide the concurrent data accession. For the example in Algorithm 1, the 4 data U, R, D, L must be passed to the computing unit in one clock cycle. However, for the stencil operations at the same iteration, the duplicated custom buffers are mainly used to provide the concurrent data accesses. The stencil data stored in these duplicated buffers are almost identical.

Rather than duplicating the custom buffer that storing the identical stencil data to increase the concurrent data accessibility. I propose an extended custom buffer design that can deliver enough stencil data to multiple computing units belonging to the same iteration with the optimal resource consumption. The detail is introduced in Chapter 4 by using the LBM simulation.

3.3.3 Computation Result Reuse Optimization

The last challenge comes from the optimization for reusing the computation result. Besides exploiting the data reusability with the custom buffer to reduce the external memory bandwidth requirement, there also exist optimization opportunities for reusing the stencil computation result in the computing units.

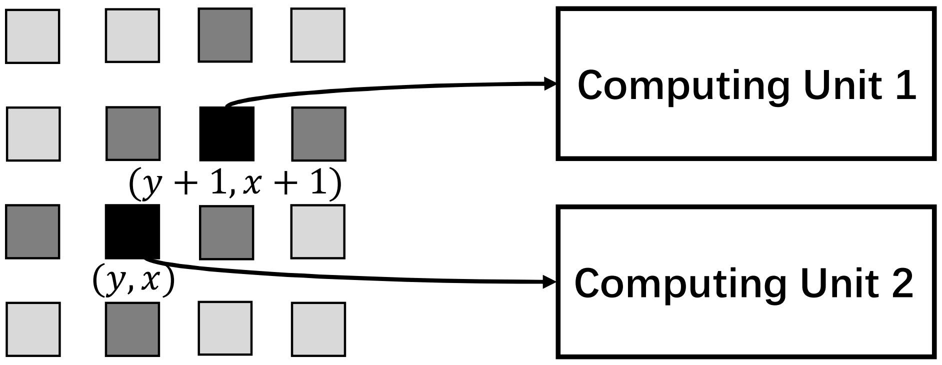


Figure 3.9: An example of the stencil computation result reuse opportunity

I also use the pseudo code in Algorithm 1 as the target stencil kernel. Figure 3.9 shows an example of 2 computation units which are responsible for performing stencil operations for two stencil cells in index (y, x) and $(y + 1, x + 1)$. For the computing unit 1, it needs 4 data from the neighbours of index $(y+1, x+1)$ to perform the stencil operation, i.e., $0.25 \times ((y+2, x+1) + (y+1, x+2) + (y+1, x) + (y, x+1))$. Similarly, the computing unit 2 also needs 4 data to perform stencil operation on index (y, x), i.e., $0.25 \times ((y+1, x) + (y, x+1) + (y-1, x) + (y, x-1))$. As a result, the computation result $(y+1, x) + (y, x+1)$ is calculated both in the computing unit 1 and computing unit 2.

Due to the flexibility of FPGA, the 2 computing units can easily merge to 1 large computing unit that perform 2 stencil operations at the same time. Thus, the calculated computation results can be shared inside the large computing unit. Meanwhile the concurrent data accesses to the custom buffer are also reduced. To benefit from that, the users need to fully explore the design possibilities in the space dimension. I explain the detail in Chapter 5.

Chapter 4

Stencil Computations for Computational Fluid Dynamics

In this chapter, I use 2 specific models of the Computational Fluid Dynamics (CFD), i.e., LGCA and LBM as the target applications of stencil computations. The CFD simulations predict and analyze fluid flows by numerically solving the conservation equations of fluid motion, which have been wildly adopted in many academic and industry areas. I first introduce the simulation background of the LGCA and LBM models. Then, I describe how I implement the parallelization strategies (spatial-based and temporal-based) to increase the performance of simulations with FPGA. For the LGCA simulation, I introduce a FIFO-based custom buffer to solve the challenge I stated in section 3.3.1. For the LBM simulation, I propose the extended custom buffer to overcome the problem in previous section 3.3.2. Finally, I evaluate the simulation architecture on the target FPGA board and compare the results with other studies.

4.1 LGCA Simulation

4.1.1 Background

Lattice gas cellular automata (LGCA) are a specific model that belongs to the family of cellular automata (CA) [58]. LGCA are popular fluid model used in bunch of applications such as in [59, 60]. The LGCA model is called HPP which is presented by Hardy, de Pazzis and Pomeau in 1973. They describe the target fluid simulation area by using the square-based lattice grid. Due to the macroscopic limitation, the HPP model can not perfectly satisfy the conservation equations, i.e., Navier-stokes. Later, they optimize the square-based HPP model to the hexagon-based FHP model [61] to solve that problem. Moreover, the unique hexagonal structure show advantage in certain applications [60]. Therefore, for the LGCA simulation, I employ the hexagon-based FHP lattice as the target model.

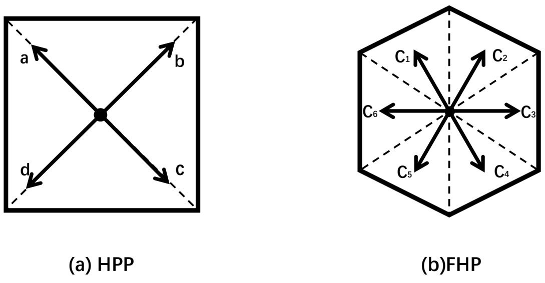


Figure 4.1: HPP and FHP lattice structure

Figure 4.1 shows the HPP and FHP lattice structure of LGCA. The movement of fluid particles is restricted inside the lattice structure. And the numerous directions of these particles' momentum are simplified to limited choices. For instance, in FHP lattice structure, the particles only can move along 6 directions, which is defined as:

$$c_i = \left(\cos(\frac{\pi}{3}(5-i)), \sin(\frac{\pi}{3}(5-i))\right), i = 1, ..., 6. \tag{4.1}$$

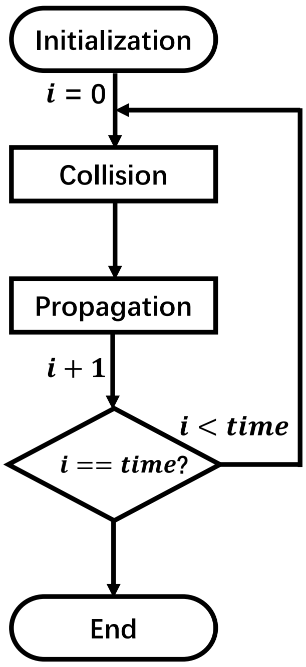


Figure 4.2: LGCA simulation procedure

The LGCA simulation procedure is shown in Figure 4.2. The procedure is mainly a loop structure which includes 2 processing stages, i.e., collision, and propagation. These 2 processing stages will repeat multiple times until the simulation reaches to the end stage. This procedure can also be represented by the equation in the following.

$$n_i(r + c_i, t + 1) - n_i(r, t) = \Delta_i \tag{4.2}$$

where $n_i(r, t)$ means the particles distribution function, the index of lattice is represented with r. t is the simulation time-step.

The propagation process is shown in the equation left side. The propagation process is used to describe how the particles of a lattice follow the corresponding momentum and move to the adjacent lattices. An example is shown in Figure 4.3 to explain this process.

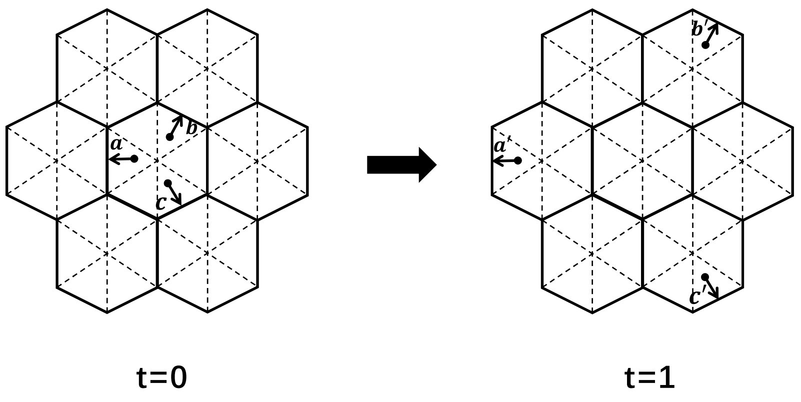


Figure 4.3: The LGCA simulation proportion example

The right side of Equation 4.2 uses the Δ_i to represent the collision process. The collision process defines how a particle collide with other particles inside the lattice. In the cases of a), g), and d) the collision results also depend on the random choice.

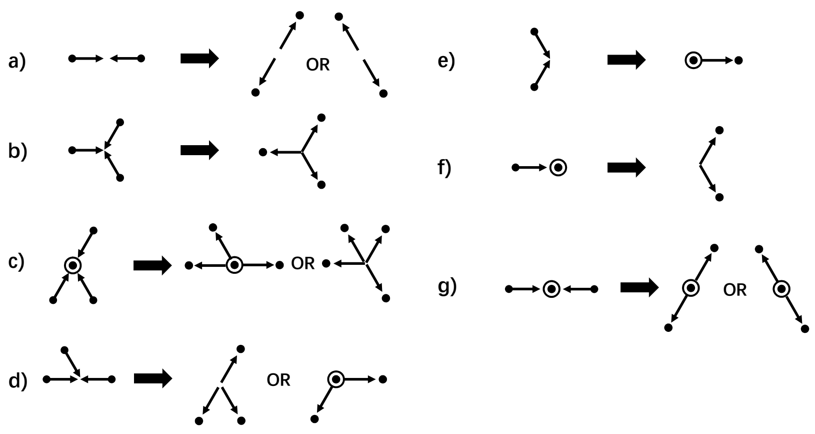


Figure 4.4: The LGCA simulation collision rules

Figure 4.4 shows some collision rules. These collision rules can use the corresponding Boolean algebra functions to detect which specific case is happened. For example, the collision rules a) and b) in Figure 4.4 can be expressed as:

$$\begin{aligned}\Delta_i = {} & [(n_i \wedge n_{i+1}) \ \& \ (n_{i+1} \wedge n_{i+2}) \ \& \ (n_{i+2} \wedge n_{i+3})] \\ & \mid [n_i \ \& \ n_{i+1} \ \& \sim (n_{i+1} \mid n_{i+2} \mid n_{i+4} \mid n_{i+5})] \\ & \mid [\xi \ \& \ n_{i+1} \ \& \ n_{i+4} \ \& \sim (n_i \mid n_{i+2} \mid n_{i+3} \mid n_{i+5})] \\ & \mid [\sim \xi \ \& \ n_{i+2} \ \& \ n_{i+5} \ \& \sim (n_i \mid n_{i+1} \mid n_{i+3} \mid n_{i+4})]\end{aligned} \tag{4.3}$$

where ξ represents the random variable to choose the 2 outputs of collision rule a).

4.1.2 Memory Storage Arrangement

In this section, I mainly describe how the lattice grid of the FHP model stores in the memory system. Due to the hexagon-based structure, there are 6 possible directions in the lattice. For each direction, a particle can either be existing or not. Besides, in the center of the lattice, there also has the position for a rest particle. Thus, in total, a lattice needs to store at least 7 bits information to express the particle distribution situations. The lattice data can be stored in a variable as shown in Figure 4.5. From MSB to LSB, each bit position corresponds to a direction.

MSB						LSB
↖	↗	→	↘	↙	←	⊙
C_1	C_2	C_3	C_4	C_5	C_6	C_0

Figure 4.5: The particles storage scheme

For the FHP model, the simulation area consists of many hexagon-based lattices. Compared to the square-based lattice model, the hexagon-based lattice grid can not directly store in the traditional memory structure. To access the target FHP lattice in the simulation area, usually the layout of the hexagon-based index need to be re-arranged to fit in the square-based index system.

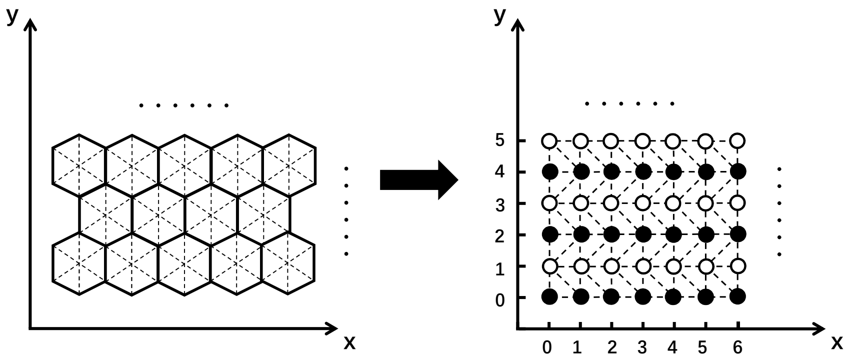
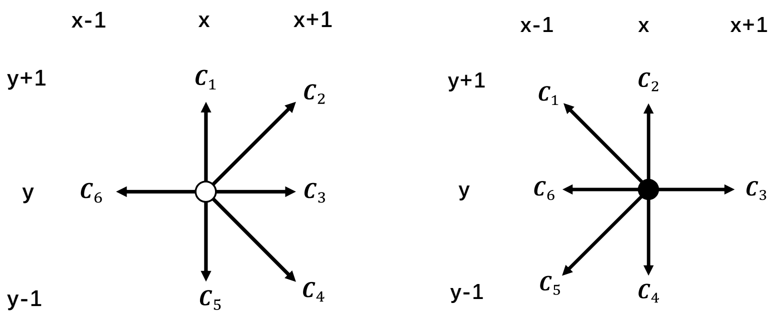


Figure 4.6: Memory layout transformation

Figure 4.6 shows the transformation detail. The hexagon-based grid is divided into two cases, i.e., white (odd lines) and black (even lines) points. They have different link structures to their neighbor lattices, which is shown in Figure 4.7. After the transformation, the lattice index r in Equation 4.2 can be changed to the coordinate like (x, y).



(a) Odd line links (b) Even line links

Figure 4.7: Hexagon-based lattice links

4.1.3 LGCA Simulation Architecture Design

The LGCA simulation process can be summarized by the following Algorithm 2. As the typical stencil applications, the Algorithm 2 mainly includes 2 kinds of loop statements. The loop in line 2 traverses the simulation time-step from $t = 0$ to the target time-step T_{end}. The loop in line 3 traverses all the lattices of the simulation area within the same time-step. These loop statements indicate LGCA simulation has good parallelism. Therefore, users can exploit the parallelization strategies that described in previous section 3.1.2, i.e., the spatial-based and temporal-based approaches to improve the simulation performance.

Algorithm 2 Lattice Gas Cellular Automata Simulation

```
1: Initialization LGCA distribution function $n_i$
2: for time $t = 0$ ; $t < T_{end}$ ; $t = t + \Delta t$ do
3:     for every lattice $(x, y)$ in the simulation grid do
4:         Calculate density and collision function $\Delta_i$ (Eq. 4.3)
5:         for every direction $c_i$ do
6:             Calculate the new distribution $n_i^{new}$ (Eq. 4.2)
7:             Streaming the particle to its neighbor
8:         end for
9:         Boundary condition check
10:    end for
11: end for
```

Architecture Design Overview

The LGCA simulation architecture overview is shown in Figure 4.8. I use the PE (processing element) for performing the collision and propagation processes as I described in the background section. One PE is responsible for processing the lattice simulation with the same time-step. Therefore, the spatial-based parallelization strategy is implemented inside the PE structure, i.e. one PE can process multiple lattices belong to the same time-step. In order to achieve that, I propose a vectorized design for the collision

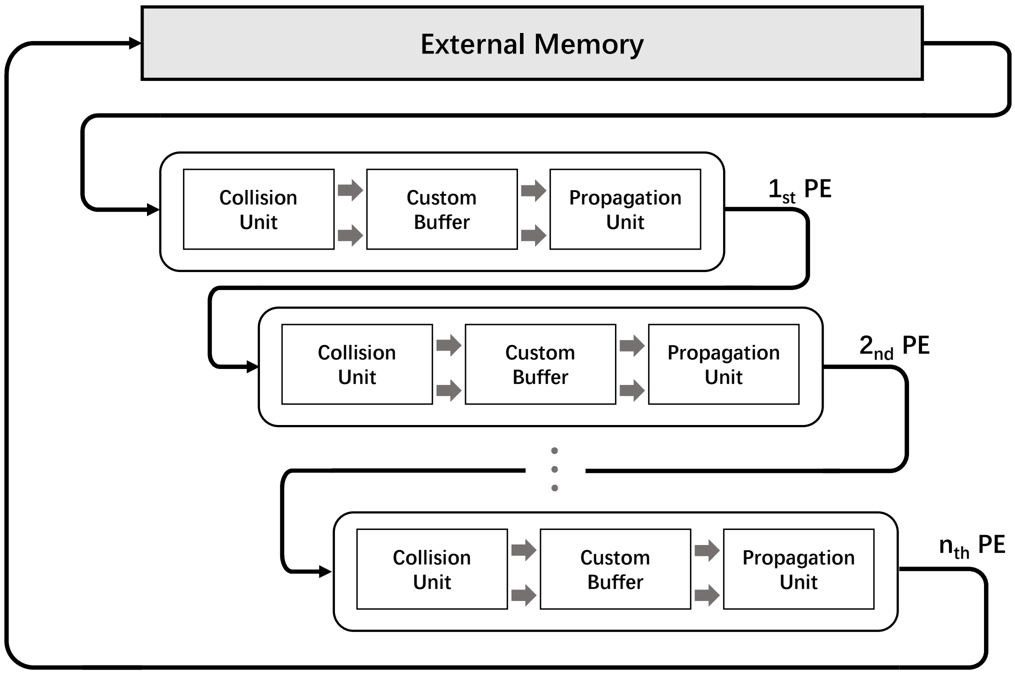


Figure 4.8: The LGCA simulation architecture overview

and propagation units. I show the details in the following section.

To scale up the simulation performance with the temporal-based parallelization strategy, I cascade the multiple PEs with different time-steps to build a chain structure. Only the 1_{st} and the n_{th} PEs accesses data from the external memory. The other PEs reuse the simulation results from previous time-step PEs, which means the number of PEs can increase with the constant bandwidth requirements. The custom buffer design in each PE shows significant impact on the implementation of temporal-based strategy. I explain the details in the next part.

Custom Buffer Design with Temporal-based Parallelization

As I stated in section 3.1.2, implementing the temporal-based strategy needs to consider the data dependency issue. For LGCA simulation, the propagation process needs to access data from other 6 neighbor lattices, which blocks the way to directly perform the simulation to multiple lattices across the different time-steps. The convectional methods to solve this problem is to buffer a part of relative data to the on-chip memory structure, e.g., cache.

Since FPGAs do not have the fixed cache structures, the data dependency issue can be overcame by employing the FPGA large configurable on-chip memory resources, e.g., registers, and BRAMs. Due to the flexibility of the FPGA structure, these memory resources can be used to create an application-specific custom buffer. Compared to the fixed cache structure, the specific custom buffer often have more optimization options and better efficiency.

For the LGCA simulation, the proposed custom buffer design is shown in Figure 4.9. To perform a simulation on one hexagonal lattice, such as lattice in index (x, y), it requires data from 6 neighbor lattices of the target lattice. This neighbour relationships are shown by the arrows. One thing need to be noticed here is that the lattices in black lines and the lattices in white lines have different neighbour relationships. For example, the neighbor lattice of target lattice (x, y) in the direction C_1 is $(x - 1, y + 1)$, which moves $(-1, +1)$. In the contrast, the neighbor lattice with the same direction C_1 of lattice $(x - 1, y - 1)$ is $(x - 1, y)$, which moves $(0, +1)$. To eliminate the differences, I pad 2 neighbor lattices to create a uniform custom buffer design. As a result, all the surrounding 8 neighbor lattices of the target simulation lattice need to be stored.

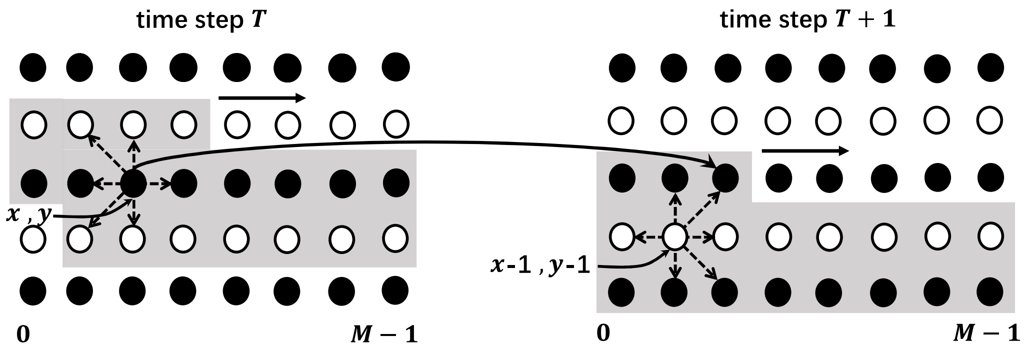


Figure 4.9: The LGCA custom buffer design

In addition to the neighbor lattices, other lattices data are also stored in the custom buffer. As shown in Figure 4.9, all lattices in the gray area are stored into the custom buffer. Thus, the total lattices in the buffer can be calculated as $2 \times M + 3$. Although these data are not needed to be directly accessed to perform the target lattice simulation, these additional data can help the custom buffer to exploit the data locality in the simulation space dimension, which reduces the external memory accessions. For example, to perform the simulation on lattice (x, y), 1 data $(x+1, y+1)$ need to be accessed from the outside. Similarly, when performing the next simulation on lattice $(x + 1, y)$, the new data $(x + 2, y + 1)$ is loaded into the buffer and the old data $(x - 1, y - 1)$ is moved out of the buffer.

For reusing data in the time dimension, the custom buffers can be cascaded in the different time-steps to form a chain structure. These custom buffers use the identical structure design. At the time-step T, after finishing the simulation of the target lattice (x, y), instead of writing the result back to the external memory, the result can be passed to the custom buffer in the time-step $T + 1$ as the new input. Therefore, When the custom buffer of time-step T loads the new input $(x+2, y+1)$ to perform simulation on $(x + 1, y)$, the custom buffer of time-step $T + 1$ can also perform the simulation on $(x - 1, y - 1)$ in parallel.

As I stated in the section 3.3.1, for HLS developing method, previous researches [50] implement this type of custom buffer design by abstracting the behaviour to a shift register. Then, with the help of Intel HLS tools, these high-level abstraction can be synthesised via the dedicated shift register IP core. However, I found that their designs show limited performance under the Xilinx HLS platform. And the bottleneck comes from the custom buffer behaviour abstraction. Therefore, I propose a custom buffer design that explicitly describes the buffer behaviour by using the FIFOs and registers for computing the LGCA simulation.

Figure 4.10 shows the FIFO-based custom buffer design. Since I pad two data, the proposed custom buffer design has the same behavior regardless of the target lattice in the black or white lines. Thus, in this figure, these two line types are not distinguished. The black block M_1 is the target simulation lattice. The 8 neighbor lattices and the target lattice itself, i.e., U_0 to U_2, M_0 to M_2, D_0 to D_2 are implemented with registers. For using high-level languages, these registers can be defined as variables. The blue color represents the $FIFO_{up}$. And the $FIFO_{down}$ is shown in red color. Unlike computing devices such as CPUs or GPUs, FPGAs have dedicated hardware design for implementing FIFO structures, e.g., BRAM-based FIFO. Since software-based high-level languages usually do not have the specific FIFO data structure, HLS tools often provide the special FIFO interfaces to define the hardware FIFO structure, i.e., hls::stream with Xilinx, or ihc::stream with Intel.

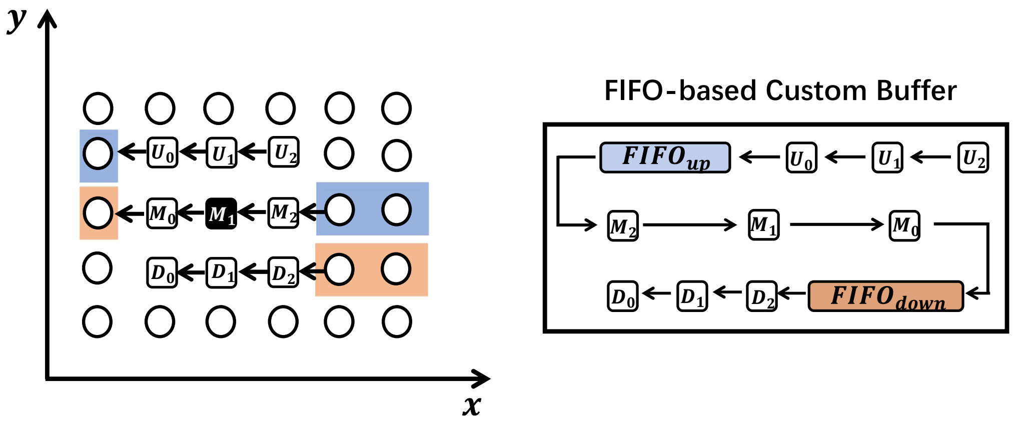


Figure 4.10: The FIFO-based custom buffer design for LGCA simulation

The behaviour of FIFO-based custom buffer is described as follows. For performing simulation on the target lattice, the register U_0 reuses data in U_1 and the register U_1 reuses data in U_2. U_2 reads the new data from external memory or another custom buffer. Data in registers M_2 and M_1 are passed to the M_1 and M_0, respectively. The new data of M_2 is popped from the $FIFO_{up}$. Similarly, the registers D_0, D_1 reuse data in D_1 and D_2. $FIFO_{down}$ pops the new data to the register D_2. At this time, all the 9 registers have been updated and ready for processing the simulation. After finishing the simulation of target lattice, data in register U_0 is pushed into $FIFO_{up}$, and data in M_0 is pushed into $FIFO_{down}$. Then, the custom buffer can repeat this data movement procedure to prepare data for the next lattice simulation.

Spatial-based Parallelization with Arbitrary Precision Data Type

In addition to cascade multiple PEs for parallel processing the LGCA simulation in time dimension, using the spatial-based parallelization strategy also can increase the simulation performance significantly. In the proposed architecture, the spatial-based parallelization is implemented inside the PE, this is, a PE which can perform the collision and propagation for multiple lattices simultaneously needs to be built.

The high flexibility of FPGAs often allows users to utilize various strategies to exploit the parallelism. For developing the simulation architecture with HLS tools, the most directly approach is to use the pragma like *pragma unroll*. Figure 4.11 shows an example. The processing unit can be duplicated easily with the unroll parameter. Meanwhile, this method also needs to increase the data throughput of the memory structure, e.g, the array a and b in this figure. The HLS compiler synthesis the array with the BRAM resource. However, the ability to access data of BRAMs in parallel in the same clock cycle is limited by the number of BRAM ports. Therefore, in order to implement the *pragma unroll*, the HLS tools usually use redundant memory resources to support concurrent memory access.

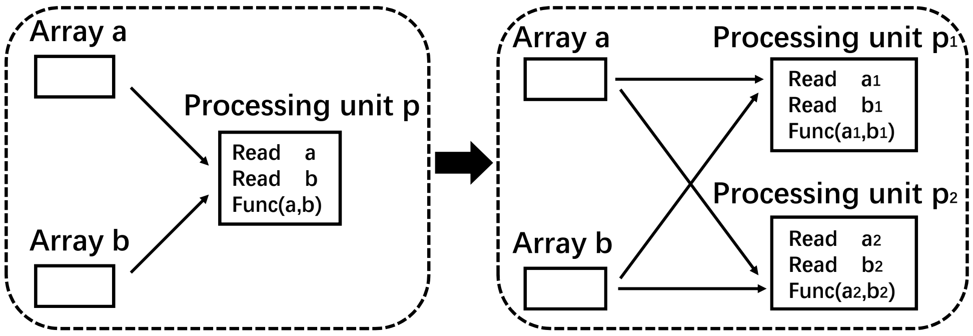


Figure 4.11: The parallelization based on *pragma unroll*

Instead of using *pragma unroll* to increase the parallelism of the PE, I propose a parallelization method based on the arbitrary precision data type. Compared with the fixed data type length of the high-level languages on the platforms such as CPUs or GPUs, the HLS tools on FPGAs have the ability to define a custom data type with arbitrary length. The Listing 4.1 show an example.

```
/* assume data type width of short is 16
int is 32, and long is 64  */
short a[512];
int b[256];
long c[128];
ap_int<128> d[64];
ap_int<512> e[16];
```

Listing 4.1: Example of using arbitrary data type

The 'ap_int <N>' is used to define a integer with arbitrary data length with Xilinx HLS environment. For example, the 'ap_int <128>' in line 6 defines a integer data type with 128-bit length. The feature of arbitrary data length is not only supported by Xilinx, it also can be used under other HLS tools such as Intel or LegUp.

In the Listing 4.1, from line 3 to line 7, these lines define 5 integer arrays with different data length. From the view of software-based programmer, they all allocate the same data space, i.e., 1KB in the memory system. However, for implementing these arrays on FPGAs, the HLS compiler can synthesis them to different memory structures. For instance, the array 'a[512]' is defined by the 16-bit integer data type 'short'. The HLS tool will store the 'short' array in a memory structure (often implemented by the BRAMs) which has 16-bit data width and 512 depth. Assume this memory structure has one port. Then, the throughput of this memory structure can provide 16 bits data in one clock cycle. To access all data in the array 'a', the 512 clock cycles are needed to complete the accession.

On the other hand, 'ap_int <512>' defines an array 'e' which stores the integer data type with 512-bit data length. Compared to the implementation of array 'a', the HLS tool synthesis the 'e' array by using the memory structure (BRAMs) that has 512-bit data width and 16 depth. Similarly, assume there is one port of this memory structure, only 16 clock cycles are needed to traverse all data in the array 'e'. By comparing the implementations of array 'a' an array 'e', although both arrays store the same amount of data, the throughput of memory structure with array 'e' is 64 times higher than the array 'a'. Hence, I can use the larger width data type to define arrays or variables to increase the throughput of the on-chip memory structure without costing redundant memory resources.

```
//Code part 1
ap_int<512> a,b,c;
c = a & b;
//Code part 2
int a[16], b[16], c[16];
for(i = 0; i < 16; i++)
  c[i] = a[i] & b[i];
```

Listing 4.2: Boolean algebra operations based on arbitrary data type

In addition to increasing the throughput of memory structure, the arbitrary data length can also be utilized to parallel execute the LGCA simulation, i.e., the collision and propagation processes. I show an example in Listing 4.2. The code part 1 performs an 'and' operation on two 512-bit data length integer variable 'a' and 'b'. The HLS tools can implement this code on FPGA in 1 clock cycle.

On the other side, the code part 2 performs the 'and' operation on two 32-bit integer array 'a' and 'b'. To complete the same task as code part 1, the code part 2 use a loop structure to repeat the 'and' operation 16 times. Compared to code part 1, the HLS tools need to cost 16 clock cycles to implement the code part 2 on FPGA. One thing need to be noticed here is this kind of parallelism can only be achieved with bitwise operations, e.g., the Boolean operations, and the shift operations.

As I stated in section 4.1.1, the collision function Δ_i of LGCA mainly consists of Boolean algebra operations. And the propagation unit can be implemented by using shift operations to move the particles data to their neighbors. Therefore, the computation capability of parallel processing collision and propagation can both be solved by performing operations between the variables with large arbitrary data type. To implement the spatial-based parallelization strategy with the proposed arbitrary data length method, I do the following changes.

- The lattice-based simulation processing units can be vectorized by using the group concept. The group utilize the data type with large length to store multiple lattices inside one group. The group size is equaled to the spatial-based parallelization parameter, i.e., $(arbitrary\ data\ length \div lattice\ data\ length)$
- Compared with storing the status of multiple lattices one by one, the group divides and stores the lattices status by the particle directions. An example of the group memory arrangement is shown in Figure 4.12,
- The memory accession with the lattice-based index system is also changed to the group-based index. Instead of performing simulation for the lattice level, the group is used as the target of the collision and propagation unit.

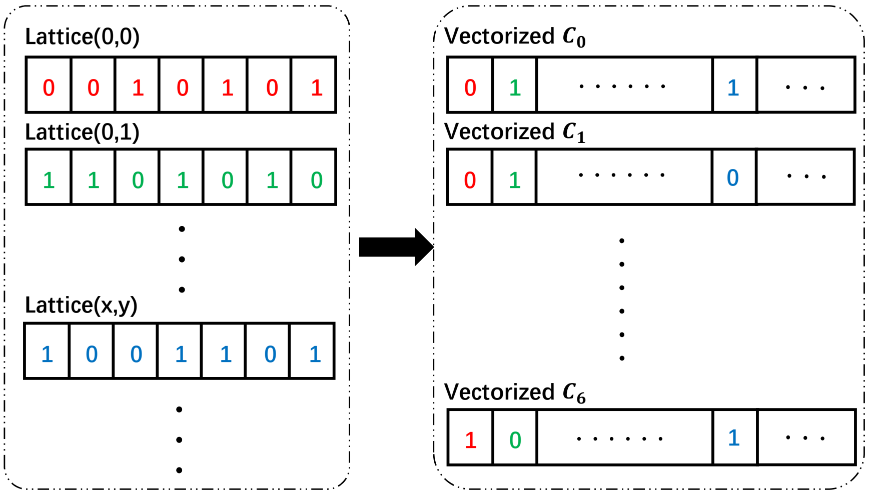


Figure 4.12: The group memory layout

A detailed example is described in the following part to clearly explain the method. The initial lattices situation is shown in Figure 4.13. For this example, I use a specific case with group size equal to 4. The index system is based on the group level. For instance, the index of the target group with the blue circle is $(1, 1)$. It includes four lattices data of which are shown in the right part of the figure. According to the

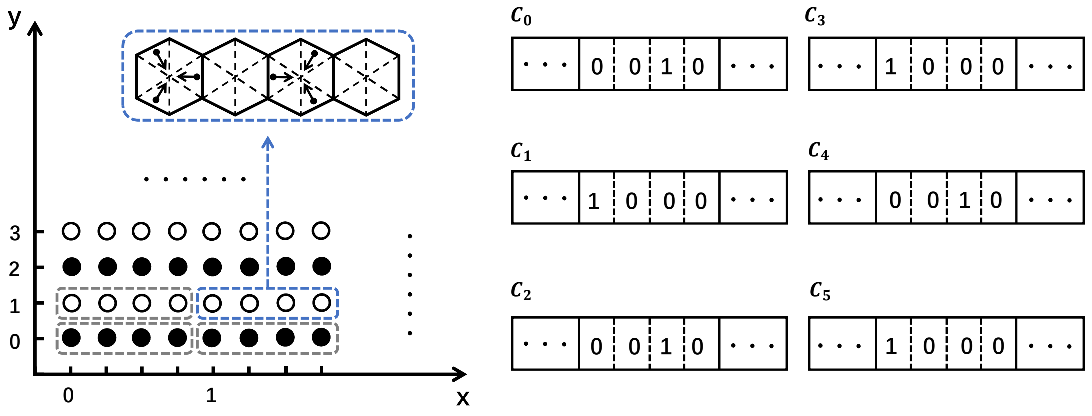


Figure 4.13: A detail example with group size equal to 4

previous description, it store these lattices information by the direction of particles C_i where $i \in 0, ..., 6$.

Assuming there is only one collision rule, i.e., the rule b) in Figure 4.4 for this specific example. The collision of the 4 lattices can be expressed by the Equation 4.4.

$$Rule = (C_i \oplus C_{i+1})\&(C_{i+1} \oplus C_{i+2})\&(C_{i+2} \oplus C_{i+3})\&(C_{i+3} \oplus C_{i+4})\&(C_{i+4} \oplus C_{i+5}) \quad (4.4)$$

To implement the group-based collision, a piece of the code example is shown in the following Listing 4.3.

```
#include<ap_int.h>
....................
#define GROUPSIZE 4
/* arbitrary data length support by hls compiler */
typedef ap_uint<GROUPSIZE> data_t;
/* the custom-sized data structure */
typedef struct group_type
{
data_t c0;
data_t c1;
....................
data_t c6;
}group;
....................
/* collision processing */
group a = local_storage[1][1];
....................
three_collision = (a.c0^a.c1)&(a.c1^a.c2)&
(a.c2^a.c3)&(a.c3^a.c4)&(a.c4^a.c5);
....................
/* update the data base on collision rules */
a.c0 = a.c0^three_collision;
....................
```

Listing 4.3: Group-based collision computation

To implement the group-based propagation, the situation is divided into two categories: black lines propagate to white lines and white lines propagate to black lines. For the purpose of simplicity, only the propagation of particles in direction C_0 will be used in this part. Figure 4.14 shows the propagation example.

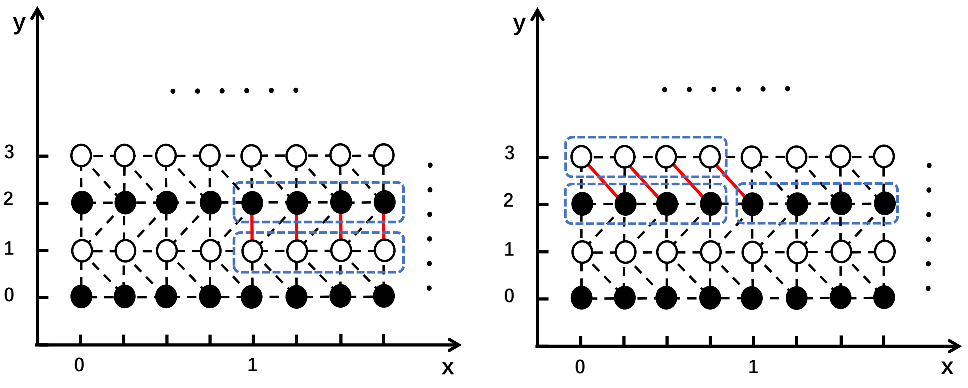


Figure 4.14: The group-based propagation example

- White to black propagation. For example, particles in C_0 direction of group $(1, 1)$ are propagated to group $(1, 2)$. This type propagation need to read data in group (1,1) and write to the group $(1, 2)$. The left side of Figure 4.14 shows this situation.
- Black to white propagation. The particles of C_0 direction in group $(0, 3)$ need to combine 3 bits data from group $(0, 2)$ and 1 bit data from group $(1, 2)$. This type propagation first reads data from the 2 group $(0, 2)$ and $(1, 2)$. Then, combining the data of target group $(0, 3)$ by shifting data in group $(0, 2)$ 1 bit to the left and shifting 3 bits of $(1, 2)$ to the right. At last, writing the combined data into $(0, 3)$. This situation is shown in right side of Figure 4.14.

According to above description, both the collision and propagation processes of LGCA simulation can be vectorized by using the group concept. As a result, I could realize the spatial-based parallelization of LGCA simulation inside the PE with different group size.

Performance Analysis

To increase the performance of proposed LGCA simulation architecture, I utilize parallelization strategies both from temporal-based and spatial-based. In this part, I will discuss how these strategies impact the simulation performance by building a performance model. This performance model can guide us to choose the optimal combinations of these strategies for the specific FPGA board. To calculate the performance, the total workload of LGCA simulation W_{LGCA} is defined by:

$$W_{LGCA} = i \cdot (M \times N) \tag{4.5}$$

where $M \times N$ means the target 2D LGCA simulation array size, i represents the total time-steps of the simulation.

Assume the target simulation architecture generates the results without stalling, i.e., for ever clock cycle I can write the result back to the external memory. Thus, my proposed simulation architecture throughput can be calculated as:

$$th_{LGCA} = f_{kernel} \times p_{PE} \times n \tag{4.6}$$

Where the f_{kernel} means the target FPGA operating frequency. The number n represents how many PEs in the simulation architecture, i.e., the value of temporal parallelism. p_{PE} means the parallelism value inside the PE, i.e., the spatial parallelism. As a result, the simulation throughput th_{LGCA} mainly depends on both parameters of p_{PE} and n.

However, the Equation 4.6 only shows the idea situation which omits the initial latency of the simulation architecture. The initial latency can be calculated from the time difference between the simulation architecture reads its first input from the external memory and writes the first output back to the external memory. Thus, the LGCA simulation time including the latency is defined by:

$$t = \frac{W_{LGCA}}{th_{LGCA}} + n \cdot (\frac{S_{PE}}{f_{kernel}}) \tag{4.7}$$

where S_{PE} represents the stage number of the pipeline system inside the PE which mainly contains the latency of collision processing, custom buffer data movement and propagation processing.

For the simulation with large workload L_{total}, the stages of pipeline latency S_{PE} often can be ignored. From the above analysis (Equation 4.7), I can reduce the total simulation time by using larger value of p_{PE} and n. However, the maximum value of these 2 parameters are limited by the target FPGA hardware resources. Let R_{MAX} denote the maximum on-chip resources of FPGA and B_{MAX} denote the maximum external memory bandwidth. The restrictions of the value n and p_{PE} are shown in the follows:

$$n \cdot R_{PE} + R_{platform} \leq R_{MAX} \tag{4.8}$$

$$p_{PE} \propto R_{PE} \tag{4.9}$$

As shown in Equation 4.9, the resource consumption of PE R_{PE} has a linear relationship with the value of p_{PE}. Increasing the number of PEs n also needs to consider the resource limitations R_{MAX}. Compared with the limitations of value n, the value of p_{PE} is constrained by the external bandwidth B_{MAX} :

$$p_{PE} \times f_{kernel} \times W_{lattice} \leq B_{MAX} \tag{4.10}$$

where $W_{lattice}$ means the data length of the LGCA lattice.

4.1.4 Experimental Results

Experiment Setup

The LGCA simulation architecture is implemented with the Xilinx VCU1525 FPGA board. This board uses the XCVU9P-L2FSGD2104E as the FPGA chip. The chip specifications is shown in Table 4.1. The VCU1525 board uses 4 16GB DDR4 2400Mhz DRAM banks as the external memory. However, I only uses 2 banks due to the limitation of the inter die connection. The target FPGA board is installed on a Intel Xeon Gold based server which has 96GB memory. The server uses CentOS 7.4 as operating system. The HLS developing method is based on Xilinx SDAccel 2018.2.

Table 4.1: VCU1525 chip specifications.

Device	Logic Cells(K)	DSP Slices	Memory (Mb)	I/O
XCVU9P	2586	6840	345.9	676

The FPGA implementation overview is shown in Figure 4.15. The host server is responsible for initializing the LGCA simulation data. With the help of SDAccel platform, the simulation data are copied to the FPGA off-chip memory, i.e., the external memory through the PCIe bus. The external memory and FPGA on-chip memory are connected by the AXI interface.

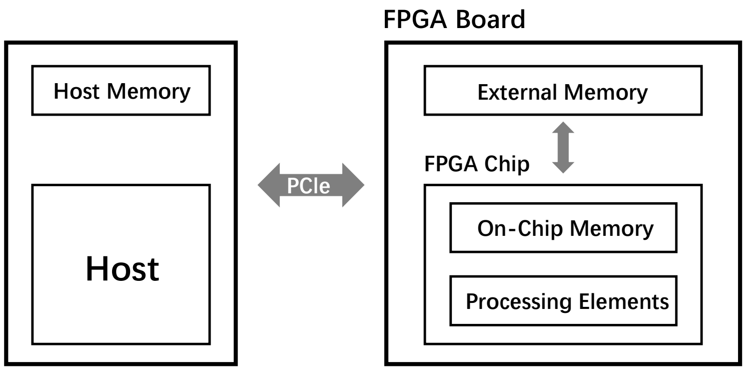


Figure 4.15: The target FPGA experiment system

For the CPU-based implementation, the Intel i7-6700 processor is used as the target comparison device. The CPU is connected with 4 32GB DDR4 2133Mhz DRAM banks. Ubuntu 18.04 is the target operating system. The software-based compiler is the GCC 7.3.0. The default compiler options OpenMP is chosen to support the multi-thread programming.

The GPU-based implementation is based on the Nvidia Quadro P5000 GPU. The target GPU has 2560 CUDA cores. A 16GB GDDR5 SDRAM bank is used as the external memory. The developing tool is the CUDA 10.0. The target GPU is installed in the previous CPU-based platforms.

Results and Comparison

The LGCA simulation experiments work with the simulation area that contains (2048×4096) lattices. The target collision rule set is the FHP-III which includes 76 particles colliding situations. The simulation results is represented by using the Million Lattice Updates Per Second (MLUPS). The definition is shown as follows:

$$P_{MLUPS} = \frac{W_{LGCA}}{t_{LGCA}} \times 10^{-6} \tag{4.11}$$

where t_{LGCA} means the simulation time (seconds). The W_{LGCA} means the workload of LGCA.

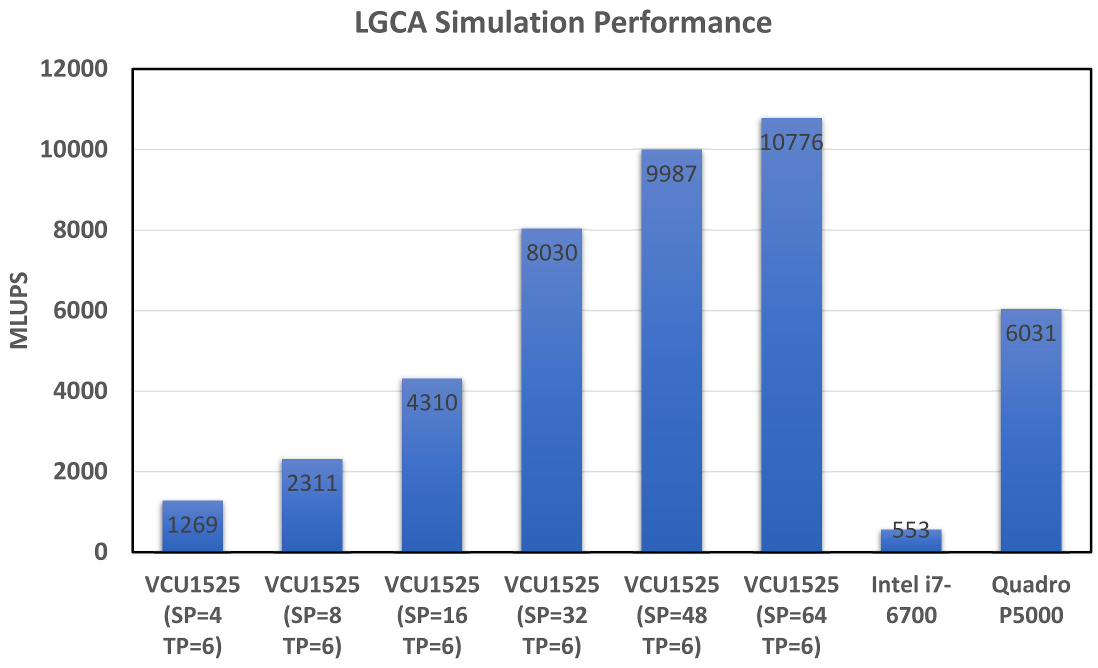


Figure 4.16: The LGCA simulation performance scales with the spatial-based parallelization

Figure 4.16 shows the LGCA simulation results. In this figure, to verify the efficiency of the spatial-based parallelization (SP) method, I keep the temporal-based parallelization value (TP) as a constant number 6, i.e., I cascade 6 PEs to implement the simulation architecture. Then, I increase the value of SP inside the PE. As a result, the total performance scales up almost linearly until the SP value reaches to 48. Compared to SP = 48, although the simulation architecture can achieve the higher performance result of 10776 MLUPS with the SP = 64, this result is heavily influenced by the external memory bandwidth. Due to the bandwidth limitation, continue to increase the value of SP will lead to inefficient hardware resource consumption. As a result, the computation architecture can not scale up the simulation performance linearly by using the SP value larger than 48.

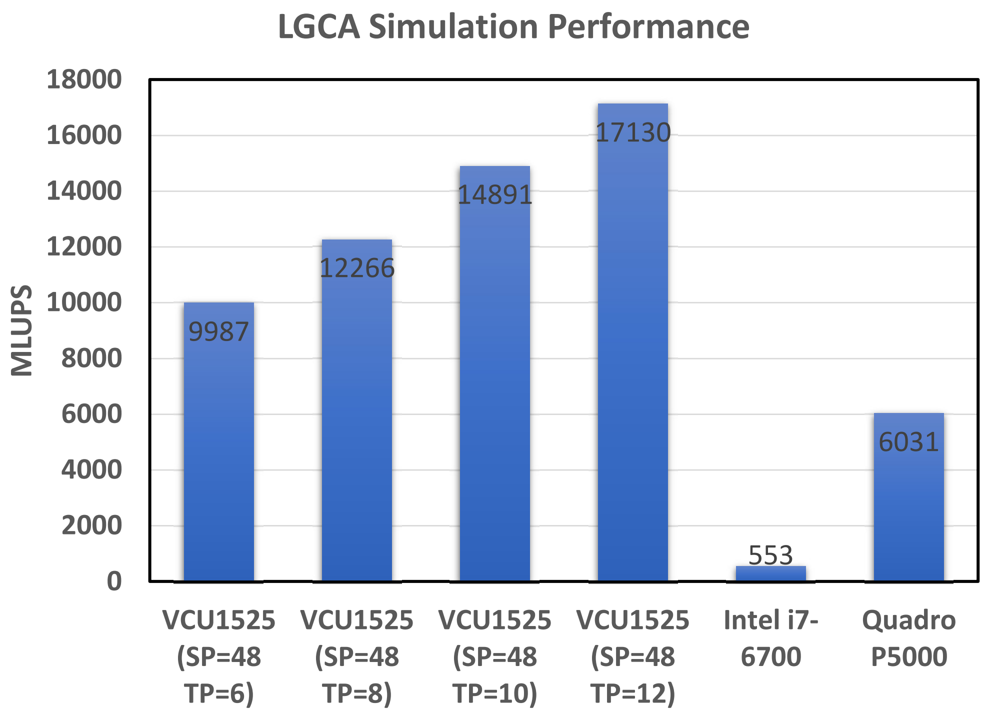


Figure 4.17: The LGCA simulation performance scales with the temporal-based parallelization

To further improve the simulation performance, I choose to employ the temporal-based parallelization strategy in the simulation architecture. Figure 4.17 shows the simulation result scale up with the value of TP. As shown in this figure, I set the SP value equal to 48 as the default parallelism inside the PE. Then, I increase the TP value from 6 to 12. The simulation performance can grow lineally with the TP value. With the TP = 12 and SP = 48, the LGCA simulation architecture achieve the highest performance 17130 MLUPS.

I implement the CPU-based design from the [58]. Moreover, I adopt the OpenMP to increase the simulation performance with the CPU multi-cores. The GPU-based design is implemented based on the work in [62]. They use the temporal blocking techniques to increase the GPU performance. The simulation results show that the simulation architecture on VCU1525 can achieve about 30 times and 3 times higher performance than the i7-6700 CPU and Quadro P5000 GPU, respectively.

Although the Quadro GPU has larger memory bandwidth, the uncoalesced memory access patterns of propagation and collision cause low bandwidth utilization. Furthermore, compared to the float-point operations, the LGCA simulation mainly uses Boolean logic operations which can not utilize the power of GPU. In addition, the large amount of brunch situations of LGCA simulation also reduce the GPU performance.

4.2 LBM Simulation

4.2.1 Background

The lattice Boltzmann method (LBM) simulation can be seen as a successor of LGCA [63]. Instead of using the Boolean algebra to describe the particles collision situations, the LBM extends the binary dynamics to the real number-based dynamics, making the LBM to be a promising approach to model more complex physical systems, especially for fluid flows.

The LBM simulation process is based on solving the Boltzmann transport equation which is shown in Equation 4.12.

$$\frac{\partial f(\vec{x},t)}{\partial t} + \vec{u} \cdot \nabla f = \Omega \tag{4.12}$$

where $f(\vec{x},t)$ means the particles distribution function (PDF). The PDF is used to express the possibilities which the particles exist in the positions $\vec{x}$ with the specific time t. The velocity of simulation particles is defined as $\vec{u}$. Ω means the collision operator.

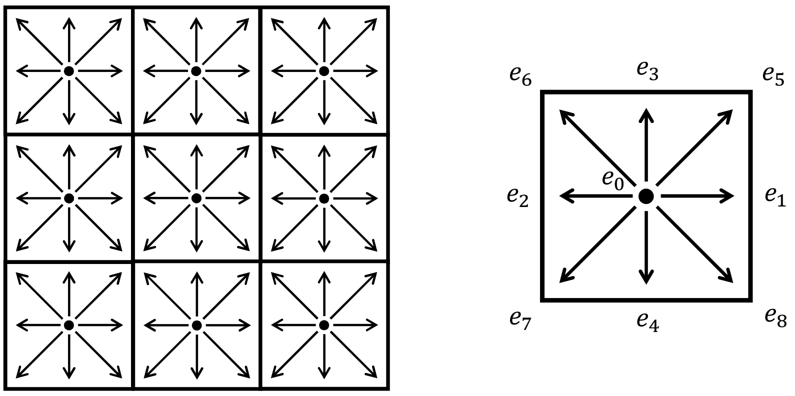


Figure 4.18: An example of D2Q9 LBM lattice grid

Like the LGCA simulation, the LBM also uses the lattice grid to simplify the target simulation area. All the particle movements are confined inside the lattices. Similarly, the simulation process is divided into the collision and propagation. A specific 2D LBM model-i.e., D2Q9 is shown in Figure 4.18.

Unlike the LGCA using a hexagon-based lattice, the LBM uses the square-based lattice grid. The particles inside the grid are allowed to propagate to 9 neighbor lattices (include the lattice itself). The relationship of these 9 lattices $\vec{e_i}$ are defined as:

$$\vec{e_i} = \begin{cases} (0,0) & i = 0 \\ (1,0), (-1,0), (0,1), (0,-1) & i = 1,2,3,4 \\ (1,1), (-1,1), (-1,-1), (1,-1) & i = 5,6,7,8 \end{cases} \tag{4.13}$$

The following partial differential equation (Equation 4.14) is used to formulate the LBM simulation process.

$$f_i(\vec{x} + \vec{e}\Delta t, t + \Delta t) - f_i(\vec{x}, t) = \frac{f_i^{eq}(\vec{x}, t) - f_i(\vec{x}, t)}{\tau} \tag{4.14}$$

where τ represents the relaxation factor which is derived from the fluid viscosity. The left part of the Equation 4.14 is used to describe the propagation of the LBM simulation and the right part represents the collision. The equilibrium distribution function f_i^{eq} is the key for computing the LBM collision rules. It can be calculated as:

$$f_i^{eq}(\vec{x}, t) = \omega_i \rho + \rho s_i(\vec{u}(\vec{x}, t)) \tag{4.15}$$

where

$$s_i(\vec{u}) = \omega_i \left[3\frac{\vec{e} \cdot \vec{u}}{c} + \frac{9}{2}\frac{(\vec{e} \cdot \vec{u})^2}{c^2} - \frac{3}{2}\frac{\vec{u} \cdot \vec{u}}{c^2} \right] \tag{4.16}$$

$$\rho(\vec{x}, t) = \sum_{i=0}^{8} f_i(\vec{x}, t), \vec{u}(\vec{x}, t) = \frac{1}{\rho}\sum_{i=0}^{8} c f_i \vec{e_i} \tag{4.17}$$

where c is the factor of time speed which is defined as $\frac{\Delta x}{\Delta t}$. The ω_i is the constant value that depends on the particle directions i. For instance, the $\omega_i = 1/36$ for $i = 5, 6, 7, 8$, $\omega_i = 1/9$ when $i = 1, 2, 3, 4$, and $\omega_i = 4/9$ when $i = 0$.

As a result, the LBM simulation process can be summarized by the Algorithm 3 as follows.

Algorithm 3 Lattice Boltzmann Method

```
1: Initialization LBM distribution function $f_i$
2: for time $t = 0$ ; $t < T_{end}$ ; $t = t + \Delta t$ do
3:     for every lattice $\vec{x}$ in the simulation grid do
4:         Calculate density $\rho$ and velocity vector $\vec{u}$ (Eq. 4.17)
5:         for every possible direction $\vec{e_i}$ do
6:             Calculate the equilibrium function $f_i^{eq}$ (Eq. 4.16 and Eq. 4.15)
7:             Calculate the new distribution $f_i^{new} = \frac{1}{\tau}(f_i^{eq} - f_i) + f_i$ (Eq. 4.14)
8:             Update the particle $f_i(\vec{x} + \vec{e}\Delta t, t + \Delta t) = f_i^{new}$
9:         end for
10:    end for
11: end for
```

4.2.2 LBM Simulation Architecture Design

Since LBM simulation is derived from the LGCA simulation, the processes of both simulation procedure are quite similar. As shown in Algorithm 3, the LBM simulation mainly includes 2 loop statements. The loop in line 2 traverses all the simulation time steps. And the loop in line 3 performs collision and propagation for every lattice in the target simulation area. To increase the performance of LBM simulation, the spatial-based and temporal-based parallelization methods are also applicable to the architecture design.

Architecture Design Overview

The overview of the LBM simulation architecture is presented in Figure 4.19. The LBM simulation architecture is mainly consists of the PEs. The PE is used to perform the collision and propagation for the lattices within the same time step. Similar as the LGCA architecture, the spatial-based parallelization is implemented inside the PEs.

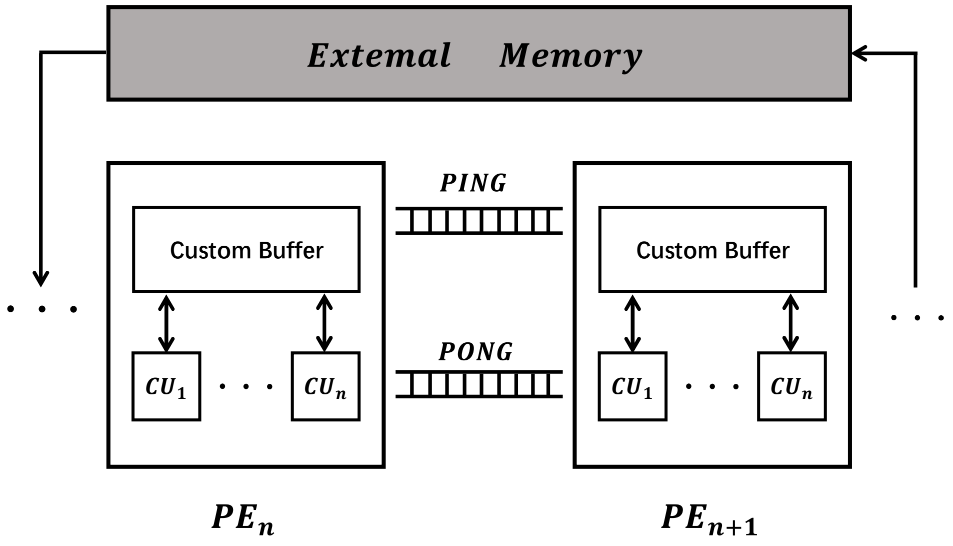


Figure 4.19: Overview of the LBM simulation architecture

To parallel execute LBM simulation for multiple lattices in one PE, I generate multiple computing units (CUs) inside the PE. The CU is the basic unit for processing the LBM simulation for 1 lattice. The CU is implemented with the one-step algorithm which fuses the separate collision and propagation of LBM simulation [64]. Instead of preforming collision with the particles inside the target lattice, the one-step algorithm chooses to pull the particles from the neighbor lattices and directly perform the collision on these lattices. All CUs inside a PE accesses data from the same custom buffer. To provide enough data bandwidth to these CUs, I need to modify the custom buffer design that proposed in section 4.1.3. I will introduce the details in the following section.

The temporal-based parallelization is implemented with the multiple PEs. These

Algorithm 4 Computing Unit Design

```
1: $e[9][2] \leftarrow \vec{e}$, $w[9] \leftarrow \omega_i$
2: for $i$ in range (0, 9) do
3:     $\rho += f[i]$
4: end for
5: $ux = (f[1] + f[5] + f[8] - (f[3] + f[6] + f[7]))/\rho$
6: $uy = (f[2] + f[5] + f[6] - (f[4] + f[7] + f[8]))/\rho$
7: for $i$ in range (0, 9) do
8:     $cu = e[i][0] \cdot ux + e[i][1] \cdot uy$
9:     $f_{eq} = w[i] \cdot \rho \cdot (1 + 3 \cdot cu + cu^2 - 1.5 \cdot (ux^2) + uy)^2$
10:    $f^{new} = f[i] \cdot (1 - \frac{1}{\tau}) + \frac{f_{eq}}{\tau}$
11: end for
```

PEs use the identical structure. Since one PE is responsible for processing lattices in one time step, I connect multiple PEs to build a chain structure that can parallel process lattices in different time steps. Similar as the LGCA architecture, only the first PE and the last PE access data from external memory. Other PEs read the inputs from the previous PEs and write the outputs to the next PEs. Thus, I can increase the total number of PEs with the constant external memory bandwidth.

Custom Buffer Design with Multiple CUs

As I stated in the LGCA simulation, an appropriate custom buffer design can not only help the PEs to overcome the data dependency issue but also utilize the data locality of target application to reduce the memory accessions of external memory. To implement the temporal-based parallelization, I can use the similar custom buffer design as I designed for the LGCA simulation.

Figure 4.20 shows that the custom buffer design adapts to the LBM simulation. The LBM lattice depends on the 8 neighbor lattices to perform the simulation. Thus, these neighbors are included in the custom buffer. Furthermore. to utilize the data locality, the other lattices covered by the gray area are also stored in the custom buffer. The temporal-based parallelization are implemented by cascading the custom buffers in sequential time steps. For example, after the simulation result of (x, y) at time step T has passed to the custom buffer with the time step $T + 1$, the next simulation in the time step T and $T + 1$ can be parallel executing. The detailed description of a similar custom buffer design with the LGCA can be found in previous section 4.1.3.

However, this kind of custom buffer design is only considered for performing simulation on 1 lattice at each time step, i.e., not exploiting the spatial parallelism of LBM simulation. Although I have proposed a spatial-based parallelization method for LGCA simulation to compute multiple lattices in parallel at each time step, this method cannot be applied to LBM simulation. For LGCA simulation, the proposed spatial-based parallelization method is based on the arbitrary precision data type, which vector-

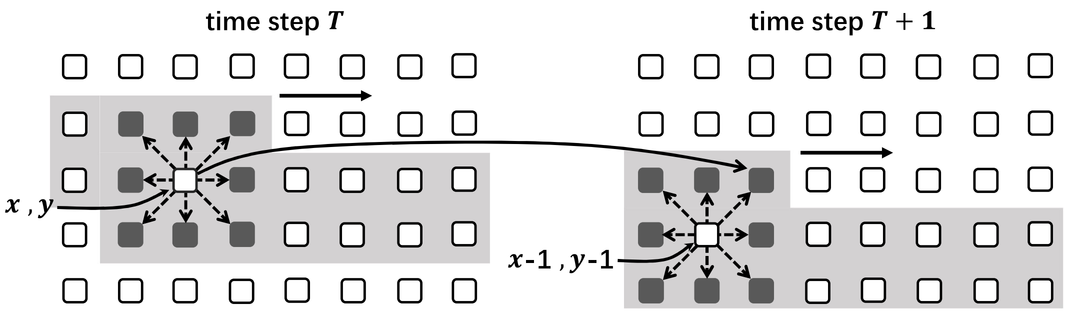


Figure 4.20: The LBM custom buffer design

izes the lattice-based computation to the group-based version. However, there exists a limitation for using the arbitrary precision-based parallelization method, that is the collision of LGCA simulation can be solved by the bitwise operations, e.g., the Boolean algebra.

Since the collision of LBM simulation is based on the real number computations, a more general spatial-based parallelization method is required. In previous section 3.3.2, I discussed the parallelization method by using the *pragma unroll*. The main problem with this method is that not only does it need to duplicate the CUs to perform simulations on multiple lattices, but it also needs to duplicate the custom buffers to provide concurrent data accessions to these replicated CUs, which costs the redundant memory resources of FPGA.

Therefore, I propose an extended custom buffer design that can increase the data throughput along with the number of CUs. The proposed custom buffer do not need to store the redundant data and also can fully exploit the data locality of the target simulation space.

Before going further, I first introduce the basic custom buffer implementation with 1 CU, which is shown in Figure 4.21. The implementation of the basic custom buffer is similar as I described in LGCA simulation. To avoid memory stalls for performing the simulation on the target lattice M_1, the CU needs to read the data of 9 lattices which includes the 8 neighbors and the M_1 itself in parallel. Thus, these lattices are stored in registers. Since the CU do not directly access the other lattices in the custom buffer, the other lattices are stored with the FIFOs, i.e., $FIFO_{up}$ and $FIFO_{down}$. These 2 FIFOs are responsible for updating data in the 9 registers. Since the behaviour of this custom buffer is similar as the design in Figure 4.10, I omit the detailed description here.

Assume the spatial parallelism is increased to 4, i.e., performing LBM simulation on 4 lattices inside a PE, based on the basic custom buffer design with 1 CU, an extended custom buffer design with 4 CUs is shown in Figure 4.22. Compared to the Figure 4.21, the LBM simulations on the target lattices M_1, M_2, M_3, M_4 are performed in parallel. 4 CUs are responsible for doing the computations for these lattices. To

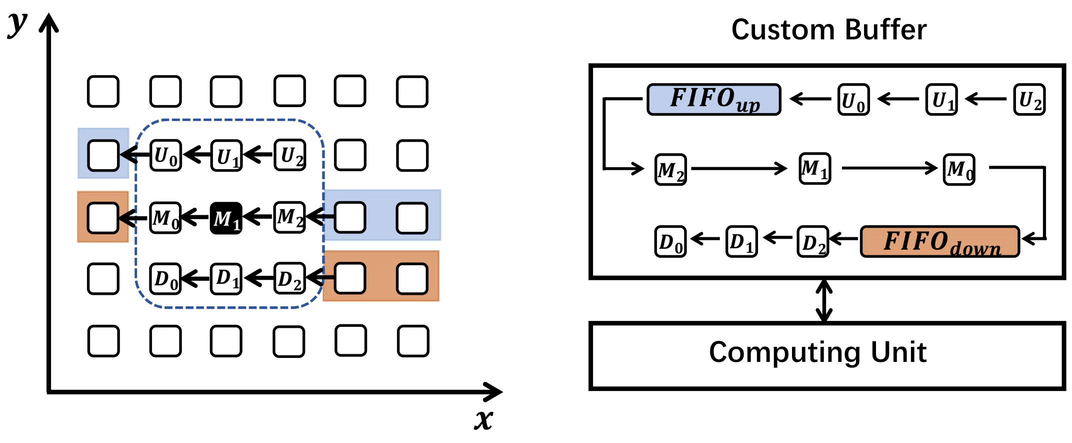


Figure 4.21: The LBM custom buffer with 1 CU

provide enough data throughput to the 4 CUs, the extended custom buffer uses 8 FIFOs and 18 registers. The behaviour of the extended custom buffer is described as follows:

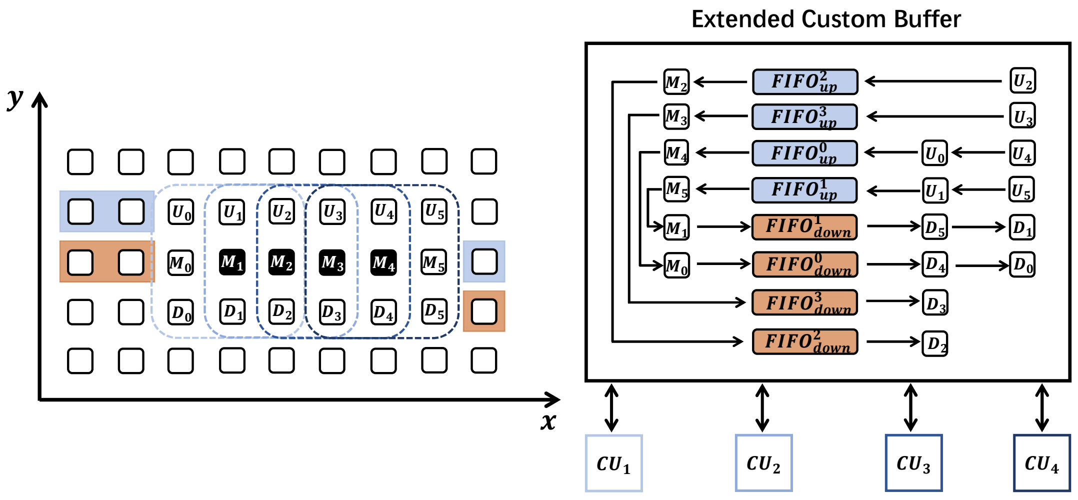


Figure 4.22: The LBM custom buffer with 4 CUs

- For performing simulation with 4 target lattices M_1, M_2, M_3, M_4, data of the 18 registers need to be prepared. the up-floor registers U_2, U_3, U_4, U_5 are responsible for inputting the lattices data outside the custom buffer, e.g., from previous time step custom buffer, or directly from the external memory. Data in register U_5 is passed to U_1. The register U_0 reuses data in U_4.
- For the middle-floor registers, the registers M_0 reuses data in M_4 and data in M_1 is transferred from M_5. Data in registers M_2, M_3, M_4, M_5 are popped from 'blue' FIFOs, i.e., $FIFO^2_{up}$, $FIFO^3_{up}$, $FIFO^0_{up}$, $FIFO^1_{up}$, respectively.
- Similar data movements are also shown in down-floor registers. The registers D_0 and D_1 reuse data in D_4 and D_5. Data in registers D_2, D_3, D_4, D_5 are popped from

'red' FIFOs, i.e., $FIFO^2_{down}$, $FIFO^3_{down}$, $FIFO^0_{down}$, $FIFO^1_{down}$, respectively. At this point, all 18 registers are ready for the 4 CUs to perform the simulation on the target lattices.

- After the CUs complete the calculations, the results of target lattices M_1, M_2, M_3, M_4, are either passed to the custom buffer of the next PE or written back to the external memory.
- At last, the 'blue' FIFOs are updated, i.e., $FIFO^0_{up}$, $FIFO^1_{up}$, $FIFO^2_{up}$, $FIFO^3_{up}$ by pushing the data in registers U_0, U_1, U_2, U_4 to the FIFOs, respectively. Also the data in M_0, M_1, M_2, M_3 are pushed into the 'red FIFOs, i.e., $FIFO^0_{down}$, $FIFO^1_{down}$, $FIFO^2_{down}$, $FIFO^3_{down}$, respectively. Then, the custom buffer can restart the same data movements to provide data to the CUs for processing the next 4 lattices.

Just like the specific example I present in Figure 4.22, This kind of custom buffer design can be easily extended along with the number of CUs. These explicitly implemented registers play an important role in providing concurrent data accesses. The number of registers is decided by the neighbor lattices of the target simulation lattices. It can be defined as:

$$n_{reg} = n_{CU} \times N_{neighbor} - (n_{CU} - 1) \times O_{neighbor} \tag{4.18}$$

where the $N_{neighbor}$ means the number of neighbor lattices which are accessed by a CU to perform the LBM simulation on 1 lattice. For example, the $N_{neighbor} = 9$ for the D2Q9 LBM simulation model. The n_{CU} represents the number of CUs that running in parallel.

Since some neighbor lattices can be shared with the multiple CUs, the total number of required registers is less than the $n_{CU} \times N_{neighbor}$. For example, 9 neighbor lattices are required to process simulation on 1 lattice, however, processing simulation with 2 consecutive lattices only needs 12 neighbor lattices, which is less than $2 \times 9 = 18$ neighbor lattices. The $O_{neighbor}$ represents the overlapped neighbor lattices that are shared by the 2 consecutive lattices. In the above specific example, the $O_{neighbor} = 6$. As a result, to provide concurrent data accesses for the case in Figure 4.22, the total number of required registers n_{reg} can be calculated as $4 \times 9 - (4 - 1) \times 6 = 18$.

The FIFOs in the custom buffer design are mainly responsible for updating the registers. Although the CUs do not directly access data from the FIFOs, these FIFOs exploit the data locality of the LBM simulation to reuse data among these registers. The required FIFOs number has a linear relationship with the number of CUs. It can be calculated as:

$$n_{FIFO} = n_{CU} \times H_{LBM} \tag{4.19}$$

where the H_{LBM} means the target LBM model 'height' which is measured by the the vertical difference between the up-floor neighbor lattices to the down-floor neighbor lattices. For the D2Q9 model, the $H_{LBM} = 2$. Then, in Figure 4.22 the total required FIFOs is calculated as $n_{FIFO} = 4 \times 2$ which is equal to 8 FIFOs.

Compared with the the method that needs to duplicated the custom buffer for providing enough data bandwidth, the most advantage of the extended custom buffer design is the consumption of FPGA on-chip memory resources. To support the concurrent data accesses and exploit the data locality of 1 CU as shown in Figure 4.21, I need to store $2M + 3$ lattices in the custom buffer, where M means the row length of target simulation space.

On the other hand, in Figure 4.22, I show the extended custom buffer design that supports 4 CUs at the same time. In this case, the custom buffer stores $2M+6$ lattices. The custom buffer length l_{buffer} can be calculated by:

$$l_{buffer} = H_{LBM} \times M + (2 + n_{CU}) \tag{4.20}$$

Assume the duplicated custom buffers method is used to support the same CUs as the Figure 4.22, the $4 \times (2M + 3)$ lattices need to be stored in the on-chip memory of FPGA. Furthermore, according to the Equation 4.20, it can be seen that increasing the number of CUs has almost none impact on the extended custom buffer length l_{buffer} due to the row length M is usually far larger than the value of n_{CU}. As a result, the extended custom buffer can scale along with the number of CUs with the optimal memory resource consumption of FPGA.

Performance Model

I introduce the performance model to guide the designers to choose the suitable parallelization strategies which can achieve the optimal performance for the target LBM simulation with a certain FPGA. As I stated above, for the simulation architecture, I mainly adopt two kinds of parallelization strategies, i.e., the temporal-based and the spatial-based. Let s_{unroll} denote the number of lattices that can be parallel computed in a PE, that is the spatial-based design parameter. The t_{unroll} represents the number of PEs, that is the temporal-based parameter. Then, the total number of lattices that can be parallel processed is calculated as:

$$n_{total} = s_{unroll} \times t_{unroll} \tag{4.21}$$

$$s_{unroll} = n_{CU},\ t_{unroll} = n_{PE} \tag{4.22}$$

where the n_{total} represents the lattice number. n_{CU} is the number of CUs inside the PE. n_{PE} means the number of PEs.

Assume the pipeline structure of the target simulation architecture can run in fully speed. The theoretical peak performance of the simulation architecture can be defined as:

$$P_{peak} = n_{total} \times f_{design} \tag{4.23}$$

where the f_{design} means the target simulation architecture operating frequency on the certain FPGA. P_{peak} denotes the peak performance, i.e., the number of lattices that can be processed in 1 second with the simulation architecture.

Therefore, the total simulation time for the target simulation task can be calculated as:

$$t_{LBM} = \frac{S_{lattice}}{P_{peak}} + t_{init} \tag{4.24}$$

where

$$S_{lattice} = T_{step} \cdot (N \times M) \tag{4.25}$$

and

$$t_{init} = n_{PE} \times (\frac{C_{PE}}{f_{design}}) \tag{4.26}$$

The $S_{lattice}$ defines the total task that performs T_{step} time step simulation on $N \times M$ space area. t_{init} represents the initialize delay of the pipeline system in the simulation architecture. It can be calculated by using the C_{PE}. The C_{PE} means the clock cycles that a PE required to finish the computations.

The limitations of the n_{PE} and n_{CU} come from the target FPGA hardware resources. They can be defined as:

$$R_{design} + R_{platform} \leq R_{max} \tag{4.27}$$

$$R_{design} = n_{PE} \cdot (R_{buffer} + n_{CU} \cdot R_{CU}) \tag{4.28}$$

$$f_{design} \times n_{CU} \times W_{LBM} \leq B_{peak} \tag{4.29}$$

where the B_{peak} means the peak external memory bandwidth of the target FPGA board. R_{max} means the maximum available hardware resources of the FPGA chip, e.g., LUTs, DSPs, or BRAMs. $R_{platform}$ is the infrastructure costs in addition to the architecture design R_{design}, such as the communication interfaces with external memory. Most hardware resource consumption of R_{design} are costed by the PEs R_{PE} which mainly compose of the custom buffer R_{buffer} and the CUs R_{CU}. The number of CUs n_{CU} is also restricted by the external memory bandwidth B_{peak}.

4.2.3 Experimental Results

Experiment Setup

I evaluate the proposed LBM simulation architecture with the Xilinx VCU1525 FPGA board, the same board I used for the LGCA simulation. Since I have already introduced the board in the previous section 4.1.4, I skip the detailed description here. In addition, I upgrade the HLS developing tool as I used for LGCA simulation from Xilinx SDAccel 2018.2 to 2018.3, which allows the target design to be developed with various high-level languages such as C, C++, SystemC, or OpenCL. I choose the D2Q9 LBM-BGK model as the simulation target [63]. The target simulation area consists of (1024×2048) lattices and the particle vector is expressed in single precision.

Results and Comparison

As I analyzed in the performance model, there are 2 design parameters, i.e. the n_{PE} and n_{CU} that mainly decide the simulation architecture peak performance. From the Equation 4.28 and Equation 4.29 it can be seen that compared with the value of n_{PE}, the value of n_{CU} is limited by both FPGA hardware resources and the external memory bandwidth. However, since the LBM simulations are typical memory-intensive applications, i.e., compared to the computations, the required memory accesses of LBM simulation is high [water], the value of n_{CU} are more easily restricted by the external memory bandwidth.

For the target FPGA board, the maximum memory bandwidth of 2 banks of DDR4 2400Mhz DRAMs can be calculated as 2×19.2 GB/s. The required data bandwidth for performing simulation on 1 lattice is $W_{LBM} = (4 \times 9)$ Bytes with the single precision data type, where 4 is the data type length and 9 is the particle number inside 1 lattice. Then, according to the Equation 4.29, assume the simulation architecture is running at 250Mhz, i.e. $f_{design} = 250Mhz$, the maximum allowed value of n_{PE} can be calculated as $2 \times 250 \times 36 = 18$ GB/s ≤ 19.2 GB/s, which is 2.

The value of n_{PE} depends on the hardware resource consumption of the PE. Since a PE mainly includes a custom buffer and the corresponding CUs. Assume I only assign 1 CU to each PE, the minimum hardware resource utilization of a PE on the target FPGA VCU1525 is shown in Table 4.2. The custom buffer mainly consumes the BRAM resources, which takes 3.7% of the available BRAM resource. On the other hand, the CU costs the LUTs and DSP slices, which take 3.5% and 4.5% of total hardware resources, respectively. One thing to be noted is that the resource costs of the CU is highly related to the n_{CU}. If I assign 2 CUs for each PE, the utilization of LUTs and DSPs can be almost doubled. Among these different types, the utilization of DSPs is the highest, thereby limiting the choices of n_{PE}. Therefore, the maximum allowed value of n_{PE} can be calculated as $n_{PE} \times 4.5 \leq 100$, which is 22.

Table 4.2: The minimum resource utilization of 1 PE with 1 CU

Device	**LUT (%)**	**DSP Slices (%)**	**BRAM (%)**	**FF (%)**
XCVU9P	3.5	4.5	3.7	1.6

Figure 4.23 shows the results of the LBM simulation architecture with different

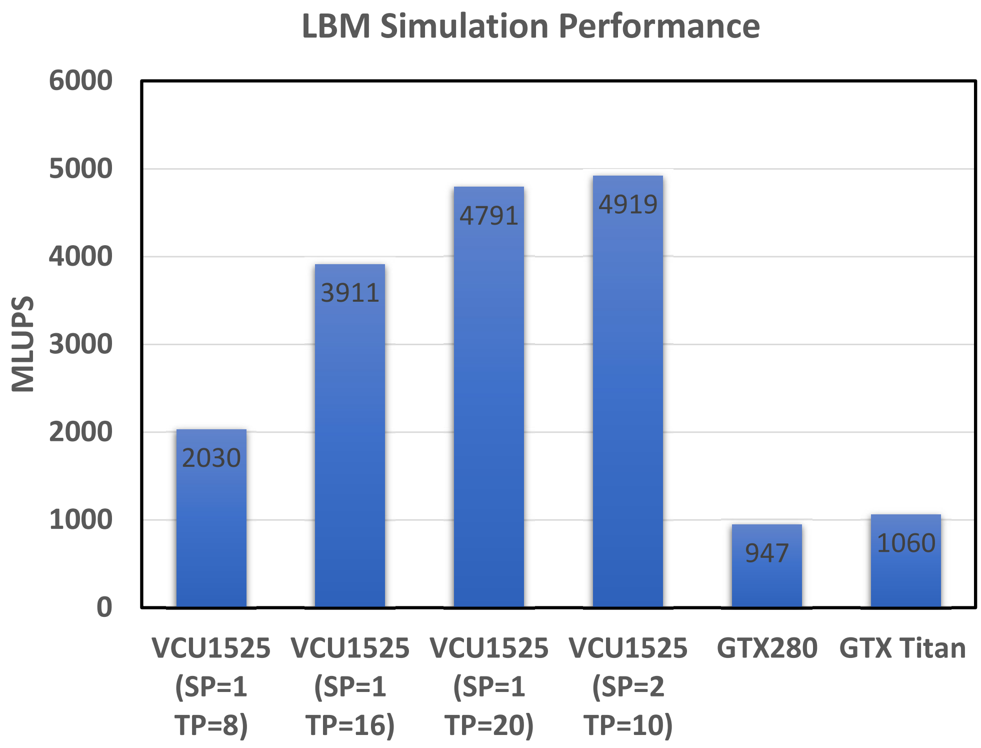


Figure 4.23: The D2Q9 LBM simulation performance

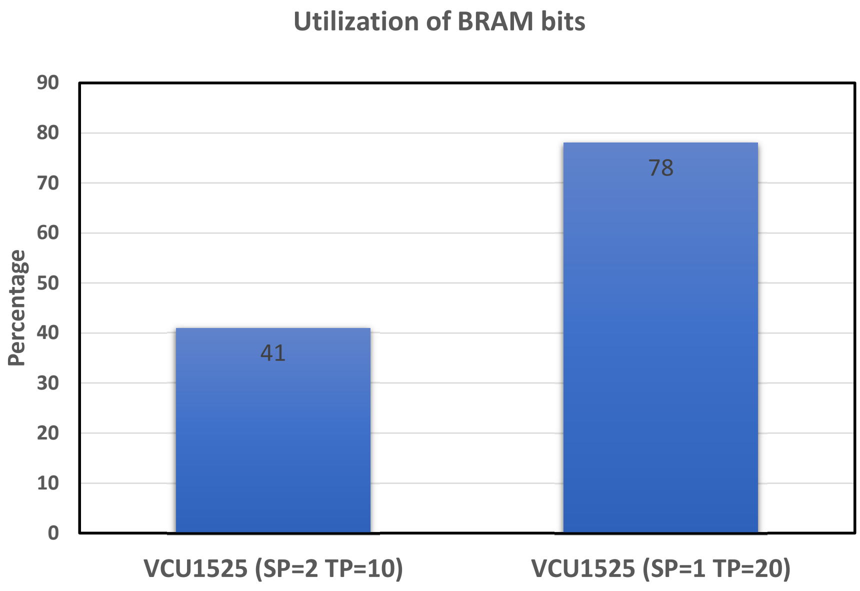


Figure 4.24: The comparison of BRAM utilization

configurations of n_{PE} and n_{CU}. The spatial-based parallelization (SP) is represented by the n_{CU}, and temporal-based parallelization value (TP) is represented by the n_{PE}. For the target simulation benchmark, the peak performance of the simulation architecture can be achieved with the $n_{CU} \times n_{PE} = 20$ which can be implemented with 2 types of architecture design. One is 20 PEs, each PE is assigned with 1 CU. The other one uses 10 PEs, each PE is assigned with 2 CUs. They can obtain almost the same level results, i.e., 4791 MLUPS and 4919 MLUPS. However, in terms of the memory resource consumption, the simulation architecture with 2 CUs in a PE ($SP = 2$) has a huge advantage, which is shown in Figure 4.24. Compared with the configuration ($SP = 2, TP = 10$), the BRAM bits utilization of ($SP = 1, TP = 20$) is almost doubled. This is mainly due to the extended custom buffer design can increase data throughput without using the redundant memory resources.

In addition to the hardware resource limitations, the peak performance of the simulation architecture is also influenced by the placement and routing process. Compared to the fixed computing devices, e.g., CPUs or GPUs, the FPGA operating frequency has a strong relationship with the routing resources that are used to connect the different hardware elements of FPGA. In case of congestion implementations, wires can be detoured, thereby causing the operation frequency to decrease dramatically or even failed to implement the design on the target FPGAs. For example, the theoretical peak performance of the simulation architecture should be obtained with the configuration such as ($TP = 22, SP = 1$). However, the implementation of this configuration will cause seriously congestion situation. As a result, the operating frequency of this implementation can be too much lower than the normal situations, which let the simulation performance even worse.

I compare the work with the GPU-based design in [65] and [66]. In the paper [65], the authors use a GPU device NVIDIA GTX Titan (Kepler architecture) to accelerate the LBM simulation. In case of D2Q9 model, their peak performance can reach to 1060 MLUPS with the double precision. For the paper [66], they use the GTX280 to implement the same D2Q9 model, they achieve the 947 MLUPS with the single precision. Compared to the FPGA devices, the external memory bandwidth of GPUs, e.g., GTX Titan with 288.3 GB/s and the float-point computing capability, e.g., GTX TITAN with 4.5 TFLOP are higher. However, the LBM simulation can not fully use these advantages. This mainly due to the following reasons: 1) similar as the LGCA simulation, the propagation and collision processes of LBM access external memory by using different patterns, making the GPUs very hard to coalesce data during the entire simulation time to exploit the wide bus width; 2) different collision rules for the lattices in boundary regions can cause branch divergence, which leads to the performance degradation.

4.3 Summary

In this chapter, I present the simulation architecture design for implementing the LGCA and LBM simulations on the target FPGA VCU1525 with the HLS developing method. For both implementations, I mainly increase the simulation performance by combining the spatial-based parallelization and temporal-based parallelization strategies. The temporal-based parallelization is implemented with the multiple PEs. For using the HLS, the behaviour of PEs is described with the C language functions. I connect these functions to form a deep-pipelined structure. Inside the function, there exist 2 main code sections, i.e., custom buffer design and computing logic. I duplicate the computing logic of the LGCA and LBM simulations by using the loop unroll techniques of the target HLS compiler to realize the spatial-based parallelization. For the custom buffer, I implement the buffer by explicitly describing the data movement of the registers and FIFOs structure.

I also compared the different results of the spatial-based and temporal-based methods, showing the spatial-based method consumes less resources, especially in terms of memory resources. For LGCA simulation, I introduce a custom buffer design which explicitly describes the buffer behaviour by using the FIFOs and registers. I also propose a arbitrary precision based vectorization method to realize the spatial-based parallelization strategy. The simulation results show the simulation architecture can achieve 17130 MLUPS. For the LBM simulation, I extend the custom buffer design in LGCA to support multiple CUs without storing redundant lattice data. The best result of LBM simulation is 4919 MLUPS.

Chapter 5

Stencil Computations with High Bandwidth Memory

In previous chapter, I have already introduced the architecture design for implementing specific stencil applications such as LBM and LGCA simulations. The results show that by employing the spatial-based and temporal-based parallelization, the target FPGA can achieve the same level as the GPUs. Compared with GPUs, the external memory bandwidth of target FPGA board (VCU1525) is very low. For example, the peak performance of a DDR4 2400 DRAM bank only can provide 19.2 GB/s. On the contrary, the GDDR5 SDRAM of GPU can achieve 448 GB/s. Thus, to achieve the similar results as GPUs, the temporal-based parallelization is mainly used to scale up the performance with constant memory bandwidth. This is mainly due to the design space of spatial-based parallelization is heavily constrained by the low memory bandwidth.

In this chapter, I present the HLS-based stencil computation architecture implementations on FPGA that use the high bandwidth memory (HBM) as external memory. With the continuous development of the FPGA industry, the most advanced FPGAs have begun to be connected with the second-generation HBM, which significantly increases the external memory bandwidth. For instance, the peak bandwidth of target board Xilinx Alveo U280 used in this chapter can reach up to 460 GB/s. As a result, the HBM extends the design exploration space for target applications, especially for using the spatial-based methods and provides extra design optimization possibilities. I use 3 typical stencil applications, i.e., Sobel 2D filter, Laplace Equation, and Himeno benchmark to verify the proposed stencil computation architecture. The contributions mainly include

- I propose a stencil computation architecture by fully exploring the design space of the target applications in both spatial-based and temporal-based parallelization strategies.

- To fully explore the design possibilities and perform the computation reuse optimizations as I stated in section 3.3.3, I further extend the previous custom buffer design to multiple space dimensions.

- I show the corresponding optimization techniques to improve the utilization of HBM peak bandwidth.

5.1 Background

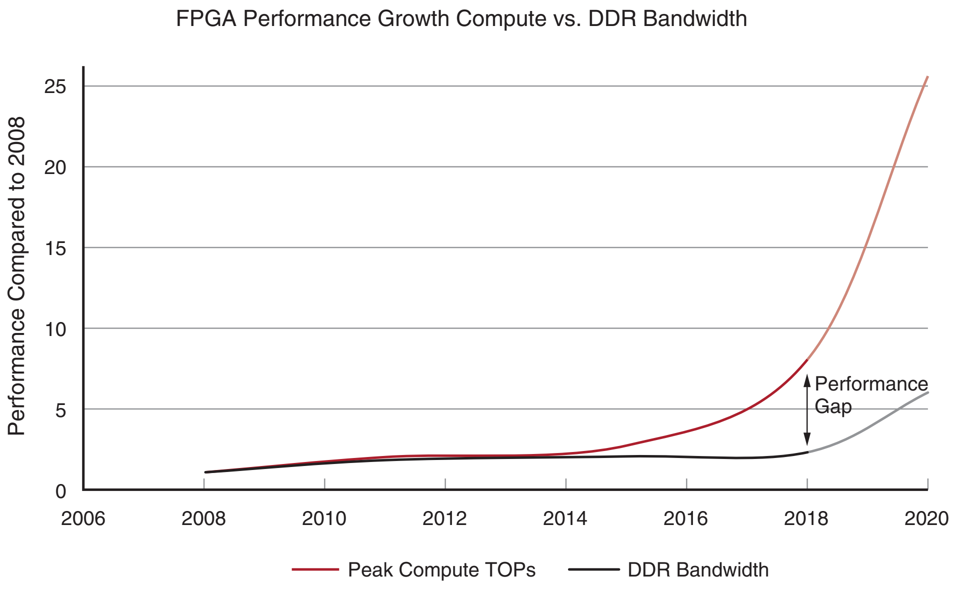


Figure 5.1: The FPGA performance growth vs. DDR Bandwidth [67]

Compared to the continuously increasing number of hardware elements inside the FPGA chips, the growth of external memory bandwidth of FPGAs is usually insignificantly, as shown in Figure 5.1. For instance, the state-of-art Xilinx Virtex UltraScale FPGA can be integrated with 12k DSP slices, which is 6 times higher than the 2k DSP slices in the largest Virtex 6 which is introduced in the year 2009. On the other side, the maximum DDR4 bandwidth supported by today's FPGAs is still less than twice that of DDR3 in 2008. As a result, connecting the FPGA chips with multiple DDR memory banks is almost the only way to significantly improve the total external memory bandwidth. However, this kind of method is usually unsustainable due to the constraints, e.g., the target FPGA chip I/O pins, total power limitations, and the cost requirement.

As the evolve of silicon packaging technology, today's FPGAs can use HBM as the external memory. Unlike the traditional DDR banks, the HBM increase the memory bandwidth by bundling the FPGA chip and the HBM DRAM chips side-by-side in the same IC package. For example, the recent standard of HBM can stack 4 or 8 DRAM chips with each others. Through stacking multiple DRAMs into the same chip package, the number of required I/O footprint can be decreased dramatically. Moreover, the close distance of the chips in the same package can help HBM achieve good performance in signal transporting.

Figure 5.2 shows the specific HBM configurations for the target FPGA board Alveo U280. There exist 2 physical HBM stacks that are integrated with the FPGA chip. For each HBM stack, there are 8 memory controllers and 1 controller can be divided into 2 channels. The channel works similarly as the traditional DRAM memory bank. The AXI3 bus protocol is used as the interface to connect the memory controllers and the FPGA chip. One thing to be noted here is that 1 memory controller only can directly access the memory space belonging to itself. For example, in Figure 5.2, 1 memory controller can manage the 256 MB data. However, for the target board U280, Xilinx has built a switch networks which let data in 1 memory controller can communicate with other controllers. Thus, the developers can have more flexibility to manage the data location.

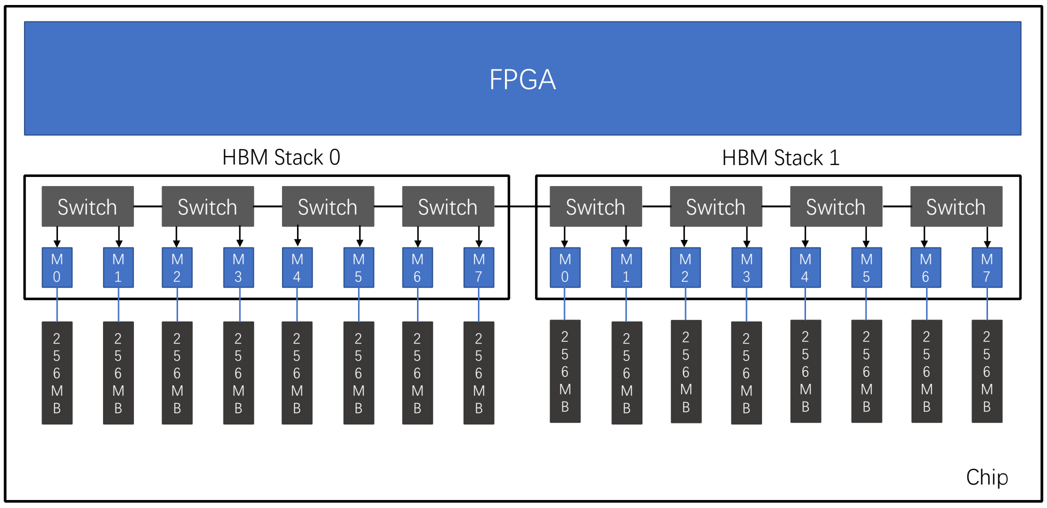


Figure 5.2: The HBM in Xilinx Alveo U280 board

5.2 Stencil Computation Architecture

In this section, I first present the the design objectives of the proposed custom buffer design. Then I describe how to extend the basic custom buffer design to fully explore the design space by using a specific stencil application. At last, I introduce the proposed computation architecture. In addition, the optimization techniques for improving the efficiency of the memory bandwidth of HBM are also discussed in this section.

5.2.1 Custom Buffer Design Objectives

Many previous studies have used the custom buffer design to build the deep pipelined system with FPGAs. In terms of stencil applications, the custom buffer can help the developers to overcome the data dependency issue when implementing the temporal-based parallelization and also exploit the data locality for reusing data between CUs. Meanwhile, I have introduced the proposed custom buffer design with the HLS developing method in the previous chapter. To the developers with high-level language design background, the design of custom buffer is important as the design of algorithm data structure. Almost all data movements and computations are based on the custom buffer design. Therefore, a suitable custom buffer design can aid the developers to obtain the optimal performance for the target stencil applications. The design objectives of the custom buffer is shown in the following:

- The custom buffer design is able to consume the minimum hardware resources to fully exploit the data reusability of the target stencil application based on the data locality.
- The custom buffer can provide enough data bandwidth for computing multiple stencil operations in parallel, which means the CU can concurrently accesses the required data to finish the stencil computations without stalls.
- With the aid of HBM external memory, the design space of the custom buffer can be extended to multiple dimensions of the target stencil data set, which can help the CUs to potentially share the computation results.

To achieve these goals, I use a specific stencil application, i.e., the 4-point Laplace Equation as the example to explain my proposed custom buffer design approaches in the following sections.

5.2.2 Custom Buffer Design Approaches

Similar as the custom buffer described in previous chapter, the custom buffer for the 4-point Laplace Equation is built with the FIFOs and registers and I explicitly describe the behaviour of the data movements between these hardware elements. Compared with the shift-register based custom buffer design (as shown in section 3.3.1), the FIFO-based custom buffer do not rely on the special shift register IP core, which can easily port to various FPGAs platforms with the HLS developing methods.

Figure 5.3 presents a PE design that includes the custom buffer and 1 CU. The CU needs to access data from the custom buffer for performing the stencil computation. The behaviour of the data movements is briefly shown as follows. Before each computation, the U register accesses data from the input of the PE. The data in registers R and D are popped from the $FIFO_1$ and $FIFO_2$, respectively. Data in register L reuses the data in C and C reuses data in L register. After the computation is complete, the data in U and L is pushed into $FIFO_1$, $FIFO_2$, respectively. The computation result of the CU is passed to the output of the PE.

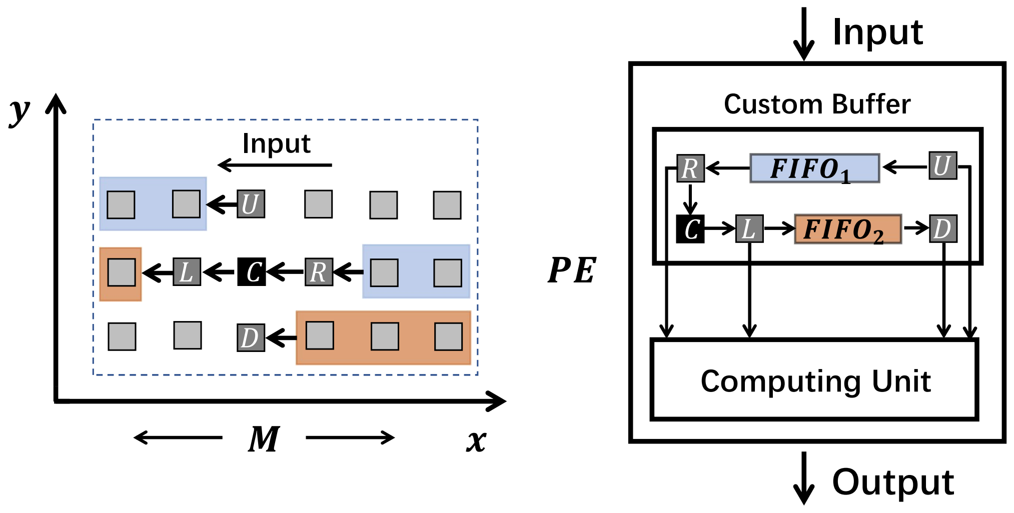


Figure 5.3: FIFO-based custom buffer design for the Laplace stencil kernel

The optimal custom buffer size to fully achieve the data reusability is can be calculated from the concept of the maximum data reuse distance that is detailed explained in [47]. For instance, data in U can be used 4 times to achieve the fully reusability, that is the data move to the registers R, L, and D. Thus, the maximum data reuse distance is the defined as the number of stencil cells between U and D, i.e., $2M + 1$ for the case in Figure 5.3. Assume I want to achieve the fully reusability of the custom buffer design, the minimum custom buffer size is equal to the maximum reuse distance, i.e, $2M + 1$.

From the analysis of the proposed stencil computation architectures for the CFD applications in the previous chapter, it can be seen that the most benefit of temporal-based parallelization is the computation performance can be scaled up without requiring extra memory bandwidth supports. Compared to employing temporal-based parallelization, spatial-based parallelization methods can also improve computing performance, but are limited by external memory bandwidth. However, in terms of resource consummations, using the spatial-based methods allow developers to exploit more design opportunities to share or reuse the hardware resources.

Due to the feature of the HBM external memory, the design space of spatial-based parallelization methods can be significantly extended. Then, the corresponding optimizations for resource sharing and reusing are also expanded. To efficiently exploit these opportunities, the custom buffer design can be scaled according to the following 3 situations. For the sake of simplicity, I explain the design detailed by 4-point Laplace Equation.

Explore Custom Buffer Design along the x Dimension of the Target Stencil Array

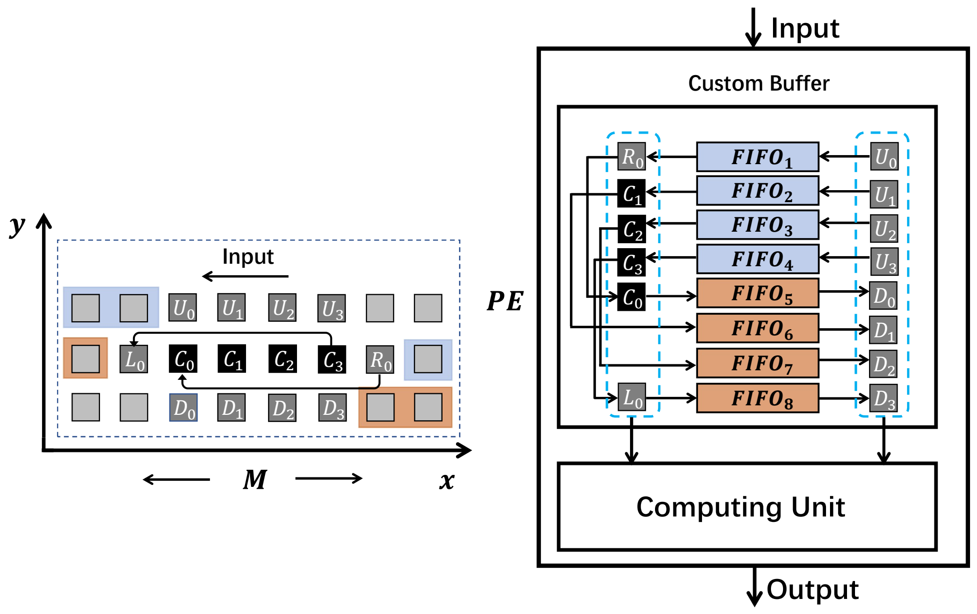


Figure 5.4: Example of a scalable custom buffer design along the stencil array x dimension.

The pseudo code for implementing the 4-point Laplace Equation is shown in Listing 3.1. As I shown in previous chapter, the most common spatial-based parallelization method is to unroll the loop statement along with the x dimension, i.e., the line 3 in Listing 3.1. For a specific situation, assume I sequentially unroll this loop by using the parameter of 4, i.e., the stencil computations are simultaneously performed on 4 sequential stencil cells in x dimension. In order to achieve the design objectives I have described in 5.2.1, the proposed custom buffer design is shown in Figure 5.4. Instead of using 1 CU for 1 stencil computations, the CU in this figure is responsible for computing 4 stencil operations. The data movements of the custom buffer is shown as follows:

- Before performing the 4 computations on the stencil cells C_0, C_1, C_2, C_3, all the required 14 registers need to be updated. The data of up floor registers U_0, U_1, U_2, U_3 are read from the inputs of the PE.
- For the middle floor registers, data in registers R_0, C_1, C_2, C_3 are popped from the $FIFO_1$, $FIFO_2$, $FIFO_3$, $FIFO_4$, respectively. The registers R_0 and C_0 is transferred to the registers C_0 and L_0.
- Down floor registers D_0, D_1, D_2, D_3 are popped from the $FIFO_5$, $FIFO_6$, $FIFO_7$, $FIFO_8$, respectively. Then, all required 14 registers are ready for the CU to perform the computations.

- After the computations, the calculated results are passed to the outputs of the PE. Data in registers U_0, U_1, U_2, U_3 are pushed in to $FIFO_1$, $FIFO_2$, $FIFO_3$, $FIFO_4$, respectively. Data in registers C_0, C_1, C_2 are pushed into $FIFO_5$, $FIFO_6$, $FIFO_7$, respectively. The register L_0 is pushed into $FIFO_8$.

I can easily generalize the custom buffer design along with other unroll parameter values. The registers in the custom buffer is the key to provide concurrently data accesses without stalls. The number of the required registers is decided by the stencil shape of the specific application and the target value of unroll parameter. It can be calculated as:

$$n_{reg} = p_{unroll} \times S_{cells} - O_{shape}(p_{unroll}) \tag{5.1}$$

where n_{reg} means the total required registers number. p_{unroll} represents the target value of unroll parameter. S_{cells} is the number of stencil cells for performing 1 stencil computations. O_{shape} denotes the stencil cells which are overlapped in the stencil shape of the sequential p_{unroll} parameter.

The FIFOs in the custom buffer are used to store and move the used data of the registers to exploit the data locality of target stencil application. To work with these registers, the minimum required FIFOs number can be calculated as:

$$n_{FIFO} = p_{unroll} \times H_{stencil} \tag{5.2}$$

where n_{FIFO} represents the number of FIFOs. $H_{stencil}$ means the 'height' of the target stencil shape, i.e., the column difference between the up floor and down floor. For the target 4-point Laplace Equation, the $H_{stencil}$ is equal to 2.

As I stated in the beginning of this section, the optimal custom buffer size to fully exploit data reusability can be calculated by the maximum data reuse distance. I can generalize the maximum reuse distance conception to fit in the proposed custom buffer design in Figure 5.4. For instance, in Figure 5.3 the maximum data reuse distance is between U and D, i.e., $2M + 1$. Similarly, in Figure 5.4, the maximum data reuse distance is between U_0 and D_3, i.e., $2M + 4$. The custom buffer size is able to define as:

$$b_{size} = H_{stencil} \times M + p_{unroll} \tag{5.3}$$

where b_{size} denotes the buffer size. M is the row length of target stencil data array. In the Figure 5.4, the buffer size is equal to $2M + 4$, i.e., the maximum reuse distance. This means that the optimal resource can be used to realize the full data reusability in the custom buffer design.

Explore Custom Buffer Design along the y Dimension of the Target Stencil Array

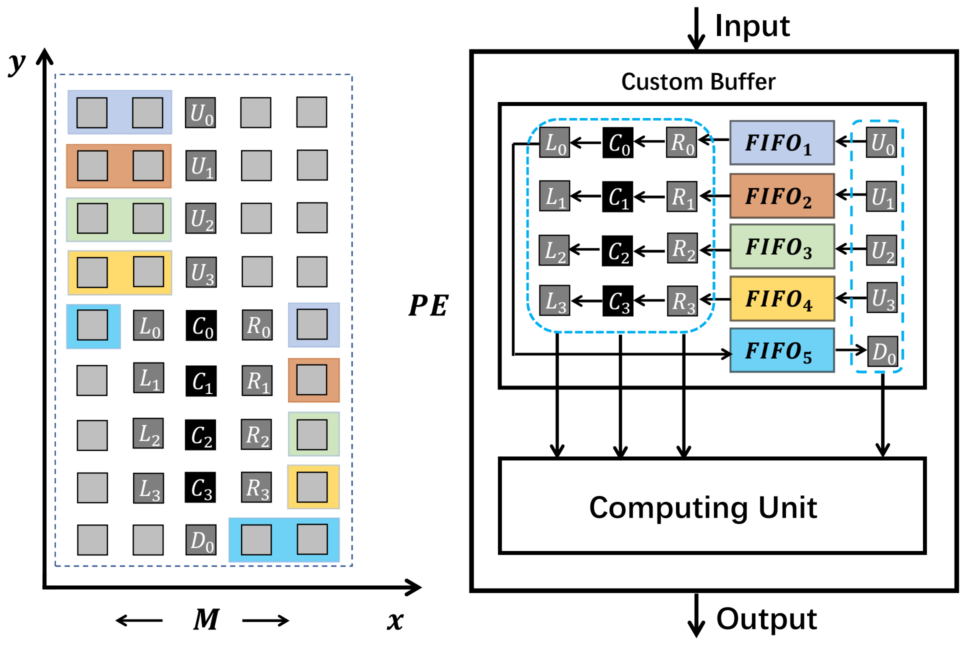


Figure 5.5: Example of a scalable custom buffer design along the stencil array y dimension.

For the target stencil array in Listing 3.1, instead of exploring the design space along the x dimension, I can also implement the spatial-based parallelization by unrolling the loop statement in line 2 that traverses stencil cells in y dimension. Still, I set the unroll parameter as 4 to illustrate the custom buffer design by exploring the design space along y dimension of the stencil array. Figure 5.5 shows the specific custom buffer design in this situation.

Due to the data movements of this custom buffer is similar as the custom buffer design in Figure 5.4, I just briefly describe the behaviour of the data movements as follows: before the computations, registers U_0 to U_3 read data from the inputs of the PE; R_0 to R_5 are popped from the $FIFO_1$ to $FIFO_4$, respectively; C_0 to C_3 reuse the data in R_0 to R_3 and pass the used data to L_0 to L_3, respectively. Register D_0 is popped from $FIFO_5$; After the computations, results go to the outputs; U_0 to U_3 is pushed into $FIFO_1$ to $FIFO_4$, respectively; L_0 is pushed into $FIFO_5$.

Compared to the custom buffer design in Figure 5.4, the custom buffer design scales along the y dimension needs to consume more hardware resources. The custom buffer size in Figure 5.5 can be calculated as:

$$b_{size} = (p_{unroll} + 1) \times M + 1 \tag{5.4}$$

Since the $p_{unroll} = 4$, the custom buffer size is equal to $5M+1$, which is significantly larger than $2M + 1$ for the design in Figure 5.4. This is mainly due to the growth of the custom buffer size has the positive proportional relationship with the unroll parameter p_{unroll}. As a result, the custom buffer resource consumption of implementing the spatial-based parallelization by exploring design space along y dimension costs more hardware resources than exploring the design space along x dimension. In addition, the custom buffer design in Figure 5.5 can be transferred to design in Figure 5.4 by converting the loop statement between the x dimension and y dimension.

Hybrid Custom Buffer Design

The hybrid custom buffer design explore the design space of the target stencil array in multiple dimensions. For the example of 2D stencil array of Laplace Equation, the spatial-based parallelization can be implemented along with both x and y dimensions. Then, the unroll parameter p_{unroll} can be defined as:

$$p_{unroll} = p_{x\_unroll} \times p_{y\_unroll} \tag{5.5}$$

where the p_{x_unroll} means the unroll parameter in the target array x dimension. Similarly, the p_{y_unroll} is the unroll parameter in the target array y dimension.

I still use the specific case $p_{unroll} = 4$ to explain the idea. According to Equation 5.5, the 3 combinations of p_{x_unroll} and p_{y_unroll} are used to implement the $p_{unroll} = 4$, i.e., $(p_{x\_unroll} = 1, p_{y\_unroll} = 4)$, $(p_{x\_unroll} = 4, p_{y\_unroll} = 1)$, and $(p_{x\_unroll} = 2, p_{y\_unroll} = 2)$. In practice, the former 2 combinations are the design methods I have introduced in the above part, i.e., along the x dimension ($p_{y\_unroll} = 1$) and along the y dimension ($p_{x\_unroll} = 1$). Therefore, I only present the custom buffer design when $(p_{x\_unroll} = 2, p_{y\_unroll} = 2)$. Figure 5.6 shows the detailed design.

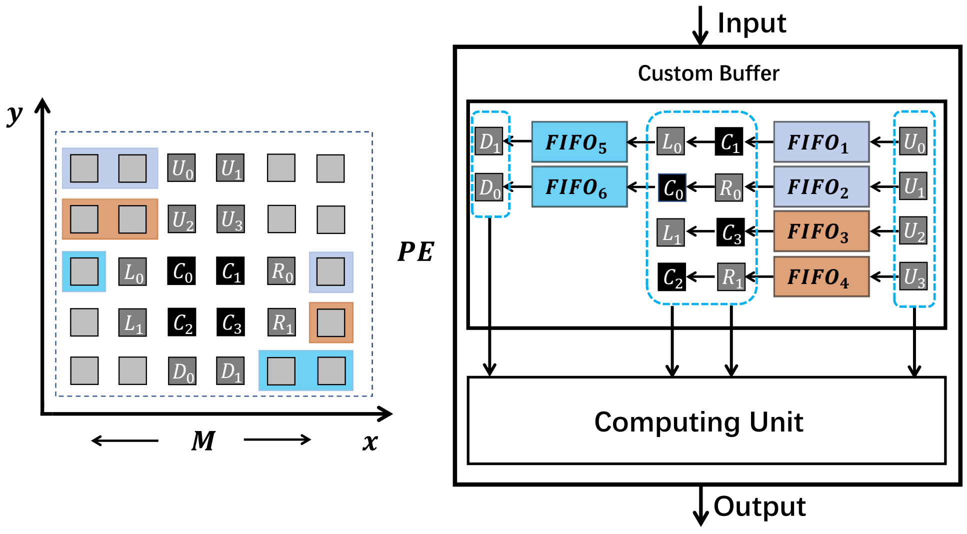

y
x
M
U_0 U_1 U_2 U_3
L_0 C_0 C_1 R_0
L_1 C_2 C_3 R_1
D_0 D_1
PE
Input
Custom Buffer
D_1 $FIFO_5$ L_0 C_1 $FIFO_1$ U_0
D_0 $FIFO_6$ C_0 R_0 $FIFO_2$ U_1
L_1 C_3 $FIFO_3$ U_2
C_2 R_1 $FIFO_4$ U_3
Computing Unit
Output

Figure 5.6: Example of a 2D hybrid custom buffer design

The behaviour of data movements is shortly descried as follows: before start the computation, data in U_0 to U_3 are access from the inputs of the PE; C_1, R_0, C_3, R_1 are popped from $FIFO_1$ to $FIFO_4$, respectively; L_0, C_0 reuse data in C_1, R_0, L_1, C_2 reuse data in C_3, R_1, respectively; D_0 and D_1 are popped from $FIFO_5$ and $FIFO_6$; when the CU finishes the computation, results are passed to the outputs of the PE; U_0 to U_3 are pushed into $FIFO_1$ to $FIFO_4$, respectively; L_0 and C_0 are pushed into $FIFO_5$ and $FIFO_6$.

From the above discussion of the custom buffer size, it can be seen that the custom buffer size is proportional to the value of unroll parameter in y dimension, i.e., p_{y_unroll}. Compared with p_{y_unroll}, the value of p_{x_unroll} almost can not impact the buffer size. Then, the custom buffer size of the hybrid design can be calculated as:

$$b_{size} = (p_{y\_unroll} + 1) \times M + p_{x\_unroll} \tag{5.6}$$

where b_{size} is equal to $3M + 1$ when $(p_{x\_unroll} = 2, p_{y\_unroll} = 2)$.

Compared to the custom buffer design in Figure 5.4, the design with $(p_{x\_unroll} = 2, p_{y\_unroll} = 2)$ needs to store more stencil cells in the custom buffer. However, as I stated in section 3.3.3, this custom buffer design opens an opportunity to make the CU share the calculated results for computing multiple stencil operations. For instance, in Figure 5.6, to perform the stencil computation on C_0, the CU has to execute the target operation, i.e., $0.25 \times (U_2 + L_0 + C_1 + C_2)$. Same as the cell C_0, to compute the stencil operations on C_3, the CU also has to calculate the result of $0.25 \times (C_1 + C_2 + R_1 + D_3)$. From the above analysis, it can be seen that, inside the CU, the computation result of $(C_1 + C_2)$ is used twice for performing stencil operations on C_0 and C_3. Therefore, by fully exploring the design space of the custom buffer design in multiple dimensions of target stencil array can provide the new opportunity for sharing the calculated results in the CU.

In conclusion, I show 3 approaches for exploring the custom buffer design space of the target 4-point Laplace Equation on a 2D stencil array. And these approaches can be easily generalized to other stencil applications with 3D or even high dimensional stencil arrays. The design space is mainly decided by the unroll parameter, i.e., the spatial parallelism p_{unroll}. Since the maximum value of the spatial parallelism is limited by the external memory bandwidth, with the aid of the HBM, the developers can fully explore the design space of the custom buffer to obtain the expected performance on the target stencil applications.

5.2.3 Proposed Architecture Overview

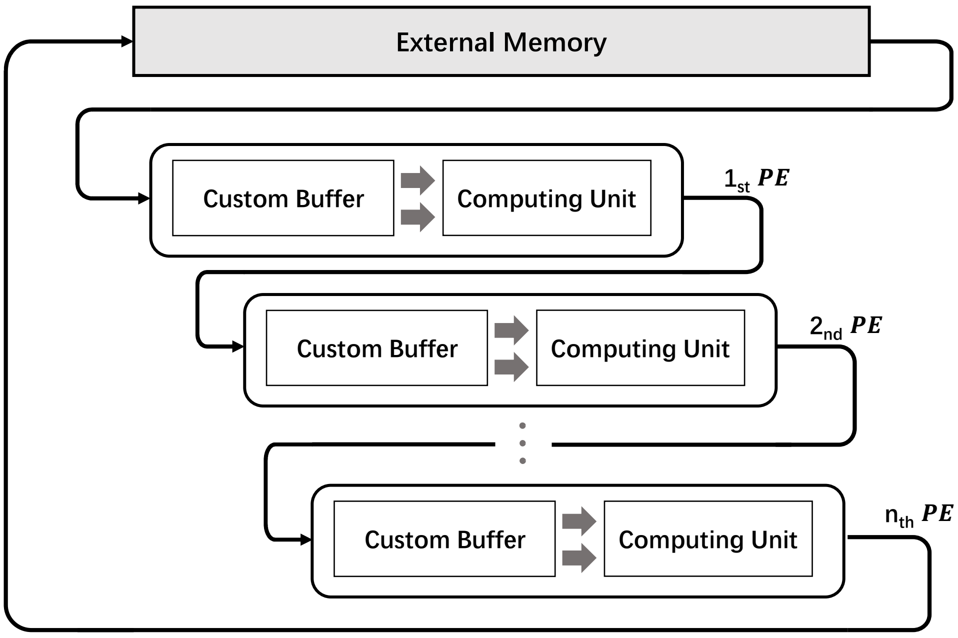


Figure 5.7: Overview of the stencil computation architecture

As the computation architecture I have proposed for the CFD applications, I also employ both temporal-based parallelization and spatial-based parallelization strategies to increase the performance of the stencil computation architecture in this section. The proposed stencil architecture overview is shown in Figure 5.7. I use multiple identical PEs to implement the temporal-based parallelization. These PEs are connected with each other to form a deep pipeline structure. Each PE is responsible for computing the stencil computations of 1 time-step. The spatial-based parallelization is implemented inside the PE. The PE mainly composes of the custom buffer and the CU.

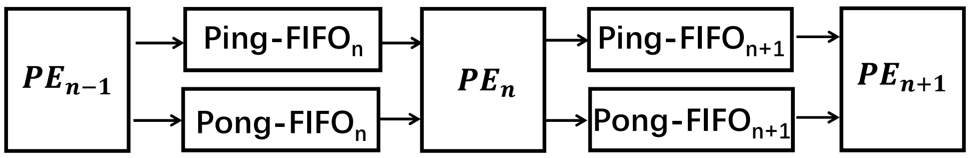


Figure 5.8: Ping-pong FIFOs connections with PEs

For using the HLS developing method, the PE can be declared as the function structure. The function interface is defined by using the FIFO data type which is not a native interface for the software-based high-level language. To avoid pipeline stalls among these PEs, I adopt the Ping-Pong FIFOs to connect these PEs. Figure 5.8 shows the details. For example, to finish 1 stencil computation, the PE reads the inputs from the previous level Ping-FIFO and writes the outputs to the next level Pong-FIFO. For the next 1 stencil computation, the PE reads inputs from the previous level Pong-FIFO and writes to the next level Ping-FIFO.

Since the Ping-Pong FIFO structure is a popular technique to enable the full pipeline system, the HLS tool of Xilinx even provide a special *pragma* to automatically implement the Ping-Pong FIFOs structure. For example, the *pragma dataflow* in Xilinx HLS can define a dataflow area. Any functions inside the area can be automatically linked with the Ping-Pong FIFOs.

5.2.4 HBM Memory Bandwidth Optimization

Unlike the fixed computing devices, e.g., CPUs or GPUs, the FPGAs do not have the predefined memory controlling unit which can efficiently manage the data accesses between the chips and external memories. For example, the memory controlling unit of GPUs can automatically coalesce the data access requests from a large number of CPU cores to increase the utilization of the external memory bandwidth of GPUs. For using FPGAs, developers usually have to manage the large amount of data requests by themselves. Thus, the unsuitable memory access patterns can cause a low utilization of external memory bandwidth.

Figure 5.9 is used as an example to show the proposed HBM memory bandwidth optimizations. Compared with the traditional developing method, using the HLS developing method often can not manage the data accesses in detail. For the target board Alveo U280, the FPGA chip can communicate with the HBM channels through AXI3 interface. To increase the utilization of HBM memory, the HLS-based memory access pattern has to satisfy the demands as follows:

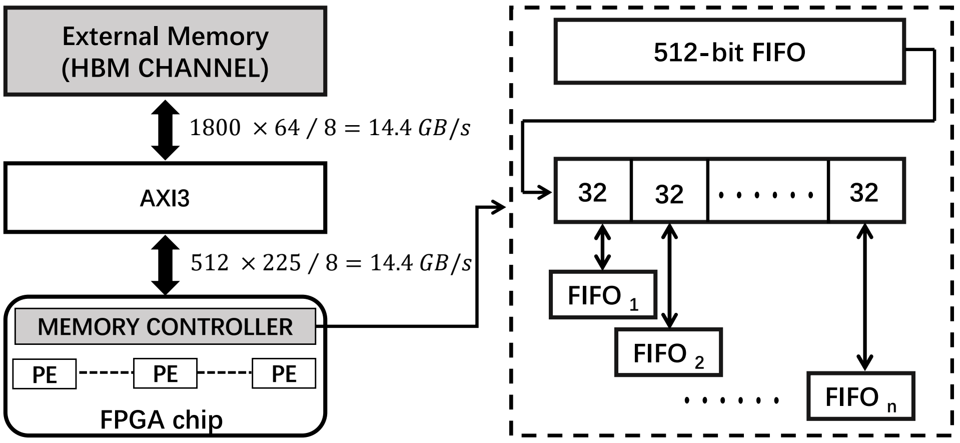

External Memory (HBM CHANNEL)
1800 × 64 / 8 = 14.4 GB/s
AXI3
512 × 225 / 8 = 14.4 GB/s
MEMORY CONTROLLER
PE
PE
PE
FPGA chip
512-bit FIFO
32
32
32
FIFO 1
FIFO 2
FIFO n

Figure 5.9: Memory bandwidth optimization

- First, due to the low operating frequency of the FPGAs, e.g., 300MHz, or even slower, the large data width of the AXI bus interface has to be used for the FPGA side. For instance, suppose the HBM memory is running at 900MHz. The theoretical bandwidth of the 1 channel of HBM memory can be calculated as 900MHz $\times$ 2 $\times$ 64 / 8 = 14.4 GB/s. To utilize the theoretical bandwidth of 1 HBM channel, suppose the target architecture design operates at 225 MHz on FPGA, the data accesses need to be explicitly merged with 512-bit width. Then, the bandwidth can achieve 512 $\times$ 225 /8 = 14.4 GB/s.

- Second, the burst mode of the AXI bus need to be exploited as much as possible.

As shown in Figure 5.9, I implement a data distribution and collection buffer to explicitly coalesce the data accesses. The behaviour of the buffer is shown as follows: when the target design needs to write to the external memory, this buffer merges the multiple data requests, e.g., represented by the data type with data width 32-bit, to a wide data width such as 512-bit; on the contrary, when reading from the external memory, the buffer split the wide data width, e.g., 512-bit into normal data type length, e.g., 32-bit. The buffer is mainly realized by a wide data width FIFO. I use the arbitrary precision integer data type to synthesis the FIFO with the certain data width. A detailed example is shown in Listing 5.1.

```
#define N 512/32
//assume the sizeof(unsigned int) is 32
//ap_uint<512> is a 512-bit width unsigned integer
hls::stream<ap_uint<512>> fifo;
unsigned int in[N], out[N];
for(int i = 0; i < N; i++)
{
 //distribution
 in[i] = fifo.range((i+1)*32-1, i*32)
 /*.................*/
 //collection
 fifo.range((i+1)*32-1, i*32) = out[i];
}
```

Listing 5.1: Custom data width FIFO structure

One thing to be noted here is the wide bit width buffer is only used to expand the AXI bus width in the FPGA side. Although I declare the buffer with the arbitrary precision integer data type, it also can fill in with float point data. For example, I can merge 16 32-bit single precision float number into the 512-bit FIFO. However, for using the high-level language such as C in the HLS tools, an extra transformation has to be done as I shown in Listing 5.2.

```
float in_float. out_float;
unsigned int in_uint, out_uint;
//uint to float, keep the binary value consistent
in_float = *(float *)&(in_uint);
//float to uint, same consistent
out_uint = *(unsigned int *)&(out_float);
```

Listing 5.2: Data type conversion between integer and float

To use the burst mode of the AXI bus, the proposed stencil architecture needs to perform the memory accesses in a pipelined manner, that is, the required data addresses must be continuous. Compared to the normal data access mode, the burst mode of AXI bus only transfer the start address and length to the receiver in the beginning, then it burst ever data without adding the address information until it reaches the target length, which increases the transmission efficiency. To access data from the target stencil array with sequential data address, I also have to linearize the data access pattern. An example is shown in Figure 5.10

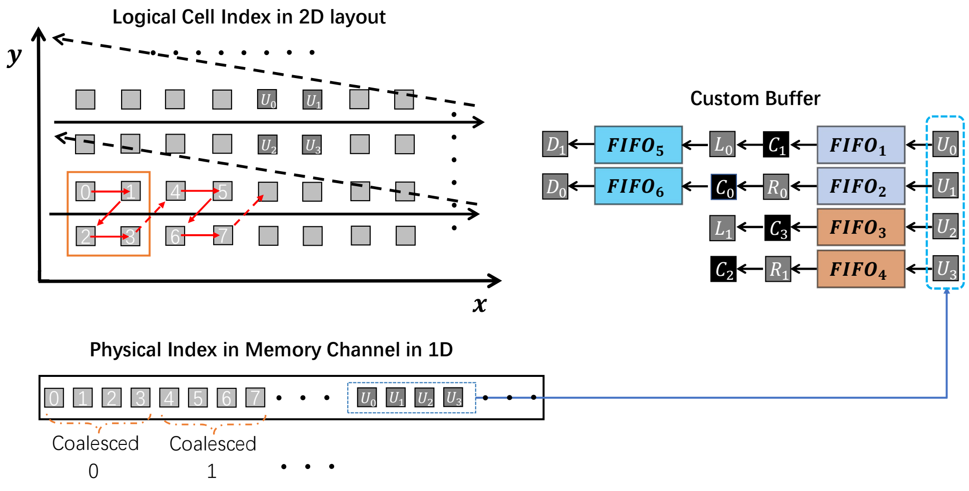


Figure 5.10: Memory access linearization

For the HLS developing method, I can use the *pragma* to configure the AXI bus burst mode. For example, accessing external memory with continuous data address in the 'for' loop can automatically trigger the burst mode. The *pragma interface m_axi* can be used to set the target length.

5.3 Performance Model

In this section, a performance mode is presented for the stencil computation architecture. By using the proposed performance model, I introduce a parameter tuning process which allows developers to choose the optimal design parameters for the target stencil application with a certain FPGA board. Since the spatial-based and temporal-based parallelization are both used in the computation architecture, the total parallelism can be calculated as:

$$p_{total} = p_{spatial} \times p_{temporal} \tag{5.7}$$

$$p_{spatial} = p_{unroll} = \prod_{k=1}^{n} p_{k\_unroll} \tag{5.8}$$

$$p_{temporal} = n_{PE} \tag{5.9}$$

where p_{total} denotes the total parallelism I have exploited in the stencil architecture. p_{unroll} is the design parameter for the spatial parallelism inside the PE, which can be explored in multiple dimension of the target stencil array. n_{PE} means the number of PEs in the stencil architecture, i.e., the temporal parallelism.

Then, I can use the value of p_{total} to calculate the peak throughput of the stencil architecture, which is shown in:

$$C_{thr} = p_{total} \times f_{stencil} \tag{5.10}$$

where the $f_{stencil}$ means the operating frequency of the stencil architecture. C_{thr} denotes the peak throughput (cells/s). The total time of the stencil architecture to execute the target stencil array can be defined as:

$$t_{stencil} = \frac{W_{total}}{C_{thr}} + T_{init} \tag{5.11}$$

$$W_{total} = i \times \prod_{k=1}^{n} D_{length}(a_k) \tag{5.12}$$

$$T_{init} = n_{PE} \times (\frac{S_{PE}}{f_{stencil}}) \tag{5.13}$$

where W_{total} is the target stencil workload for performing i time step stencil computations on an n-dimensional stencil array $\vec{a}_{(1,2,\ldots,n)}$. D_{length} denotes the dimension length of the stencil array. T_{init} represents the initial pipeline delay to generate the first result, which can be calculated by using the pipeline stages of a PE S_{PE}.

The design space of the spatial-based and temporal-based parallelization, i.e., $p_{spatial}$ and n_{PE} are mainly limited by the FPGA hardware resources which can be shown as:

$$R_{stencil} + R_{platform} \leq R_{max}(_{DSP,LUT,BRAM,FF}) \tag{5.14}$$

$$R_{stencil} = n_{PE} \cdot (R_{buffer} + R_{comp}) \tag{5.15}$$

$$R_{buffer} \propto b_{size}(p_{spatial}), R_{comp} \propto p_{spatial} \tag{5.16}$$

where $R_{max}(_{DSP,LUT,BRAM,FF})$ represents the maximum sources of the different hardware elements for the target FPGA. $R_{stencil}$ is the resource consumption of the stencil computation architecture. $R_{platform}$ denotes the platform consumption, e.g., memory interface. Inside a PE, the hardware resources are mainly consumed by the custom buffer R_{buffer} and the CU R_{comp}. The spatial parallelism design parameter $p_{spatial}$ is also constrained by the memory bandwidth of the HBM, which is shown in:

$$f_{stencil} \times p_{spatial} \times W_{datatype} \leq B_{peak} \tag{5.17}$$

where $W_{datatype}$ denotes the data width of the target stencil computation data type.

According to the previous analysis, I use a roof-line model to illustrate the design parameter tuning process of the n_{PE} and $p_{spatial}$. Figure 5.11 shows the roof-line mode of the stencil architecture. In this figure, the vertical axis denotes the performance P. The peak performance of the stencil architecture is decided by the maximum computing resources $R_{max}(_{DSP,LUT})$ in the target FPGA. The horizontal axis represents the number of PEs, i.e., n_{PE}. The $p_{spatial}$ is the slope of the lines that is limited by the blue angle B_{peak}.

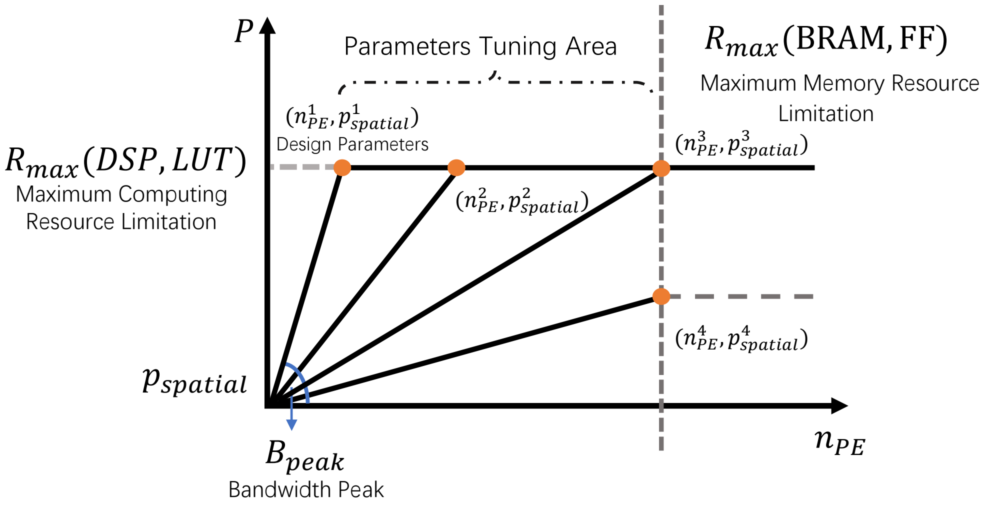

P
Parameters Tuning Area
$R_{max}(\text{BRAM, FF})$
Maximum Memory Resource Limitation
$(n_{PE}^1, p_{spatial}^1)$
Design Parameters
$R_{max}(DSP, LUT)$
Maximum Computing Resource Limitation
$(n_{PE}^2, p_{spatial}^2)$
$(n_{PE}^3, p_{spatial}^3)$
$(n_{PE}^4, p_{spatial}^4)$
$p_{spatial}$
B_{peak}
Bandwidth Peak
n_{PE}

Figure 5.11: Performance roof-line model

Then, I can conclude the tuning process of parameters n_{PE} and $p_{spatial}$ as follows:

1. Calculate the maximum value of parameter $p^1_{spatial}$ by using the peak memory bandwidth B_{peak}.
2. Keep the value of $p^1_{spatial}$ to explore the value of n^1_{PE}, until the maximum computing resource limitation is reached.
3. Add the design parameters $(n^1_{PE}, p^1_{spatial})$ into the parameter tuning area.
4. Decrease the value of $p^1_{spatial}$, e.g., to a smaller value $p^2_{spatial}$, and repeat the above steps from Step 2.
5. The parameter tuning area will end up in, e.g., point $(n^3_{PE}, p^3_{spatial})$. At this point, the maximum maximum memory resource limitation $R_{max}(_{BRAM,FF})$ is also reached.

One important thing to be noted here is using the large value of the n_{PE} may use up all the memory resources of the target FPGA before the stencil architecture reaches the peak performance. For example, the design point $(n^4_{PE}, p^4_{spatial})$ in the Figure 5.11, whose performance is restricted by the $R_{max}(_{BRAM,FF})$.

5.4 Results

5.4.1 Experiment Setup

The Xilinx Alveo U280 is used as the target FPGA board to implement the proposed stencil computation architecture. Table 5.1 shows the detailed board specifications. The FPGA chip of the Alveo U280 is integrated with 2 HBM memory stacks. Each stack of HBM can store 4GB data. The target board is connected to a host PC through PCIe interface. The host PC uses the Intel i9900k CPU with 64GB DDR4 DRAMs. The operating system of the host is Ubuntu 2018.2. The HLS developing environment is based on the Xilinx Vitis 2019.2. One thing to be noted here is that I only implement the design with 1 die of the target FPGA chip. The Alveo U280 FPGA chip includes 3 dies. These dies are connected with the super long line (SLL) routing resources. Since the target HLS developing tool, i.e., Xilinx Vitis 2019.2 has limited supports to efficiently use the SLLs, I do not consider the multiple dies implementations.

Table 5.1: FPGA chip specifications

Devices	**LUT(K)**	**FF(K)**	**DSP Slices**	**BRAMs**	**HBM Bandwidth**
Alveo U280	1303	2607	9024	4032	460.8 GB/s

I evaluate the proposed stencil computation architecture with 3 typical stencil applications from different areas. The Sobel 2D filter is a widely used method to detect the edge of the pictures. The 4-point Laplace Equation is an example for solving the partial differential equations. The Himeno benchmark is a 3D benchmark to profile the code of the fluid dynamics.

5.4.2 Experiment Performance

Figure 5.12 presents the evaluation results of the Sobel 2D filter. I set the input picture size to 8k, i.e., (8192×8192). Since the Sobel 2D filter is a special stencil application that performing the computations with 1 time step, I can not employ the temporal-based parallelization on this benchmark. As a result, the computation performance is mainly decided by the spatial parallelism, that is, restricted by the external memory bandwidth. The result of the ADM-PCIE-KU3 board comes from the paper [55]. Since their FPGA board is equipped with 2 DDR3 1600 DRAMs as the external memory, they only can implement the spatial-based parallelization (SP) up to 32, i.e., $16 \times 32 \times 250 = 12.8$ GB/s. Compared with the performance result on the Intel Xeon E5-2620, the FPGA board of ADM-PCIE-KU3 is even slower due to the memory bandwidth of the CPU is higher (4 DDR4 2133 DRAMs). As a consequence, the low external memory bandwidth prevents the FPGA board to show the advantages of parallel structures.

For the target board Alveo U280, due to the aid from the HBM memory, I can implement the spatial-based parallelization up to 384 with 16-bit data type and 640 with 8-bit data type on the stencil computation architecture. As a result, I can achieve the maximum performance results of the 16-bit in 77 GCell/s and 8-bit in 128.4 GCell/s. In addition, to show the benefits of HBM, I also compare the results with the target Alveo U280 which is connected with 2 DDR4 DRAMs as external memory. The results demonstrate that the state-of-the-art Alveo U280 board has similar performance with the Xeon CPU and the ADM-PCIE-KU3 FPGA board without the help of HBM.

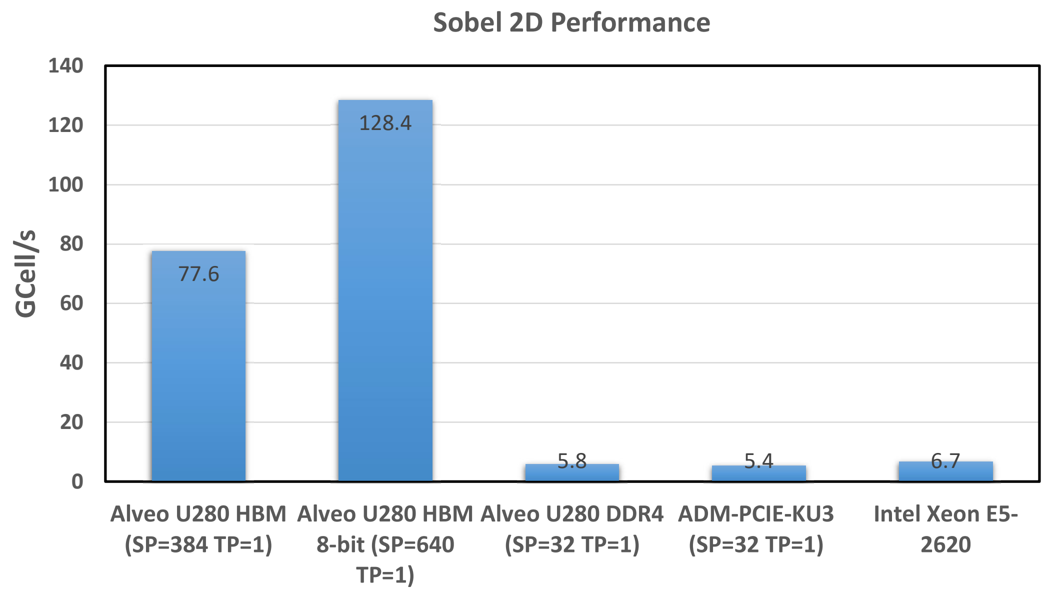


Figure 5.12: The Sobel 2D benchmark performance

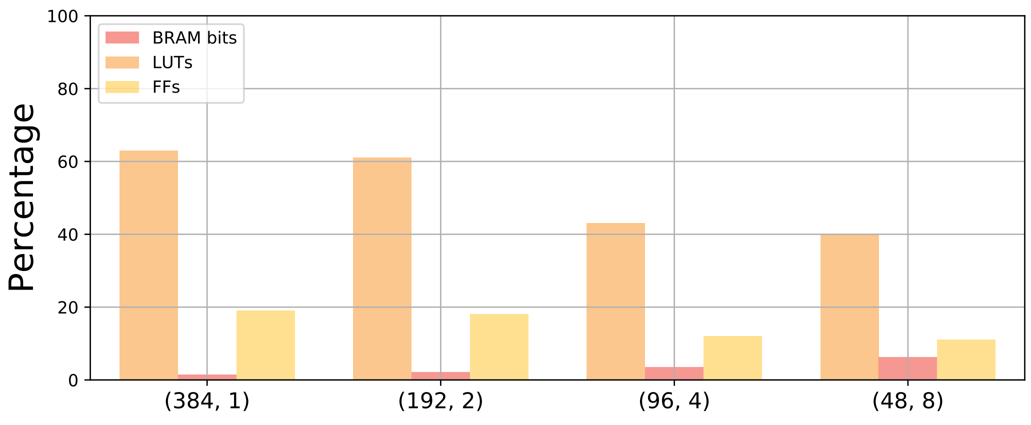


Figure 5.13: Resource consumption for the Sobel 2D benchmark

I use 4 different implementations to realize the design parameter $p_{unroll} = 384$, i.e., $(p_{x\_unroll}, p_{y\_unroll}) = (384, 1), (192, 2), (96, 4), (48, 8)$. Figure 5.13 presents the resource utilization of these 4 implementations. Since some computation results can be shared in the CU as shown in Figure 5.14, the resource utilization of the 4 implementations are also variant. For example, the configurations of (48, 8) uses the least computing resources, i.e., LUTs. On the contrary, the (384, 1) costs the minimum memory resources such as BRAMs of the target FPGA.

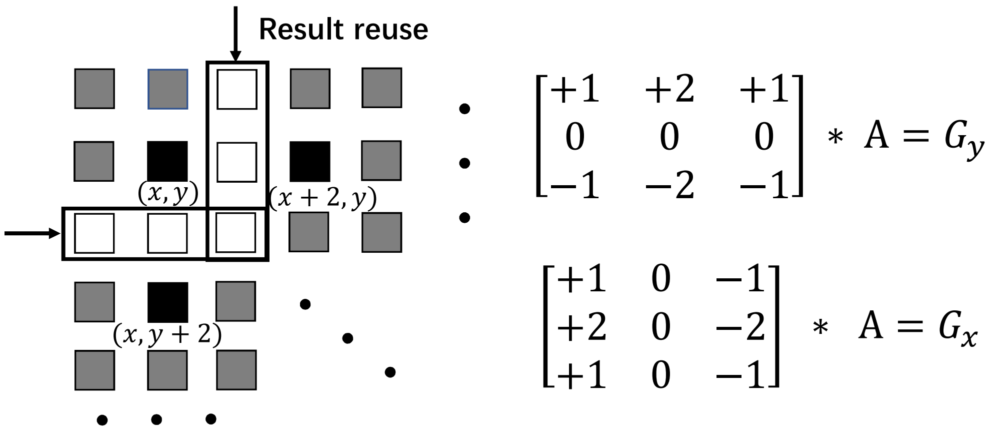

Result reuse
(x, y)
$(x+2, y)$
$(x, y+2)$
$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A = G_y$
$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A = G_x$

Figure 5.14: Computing results sharing in the CU of the Sobel 2D benchmark

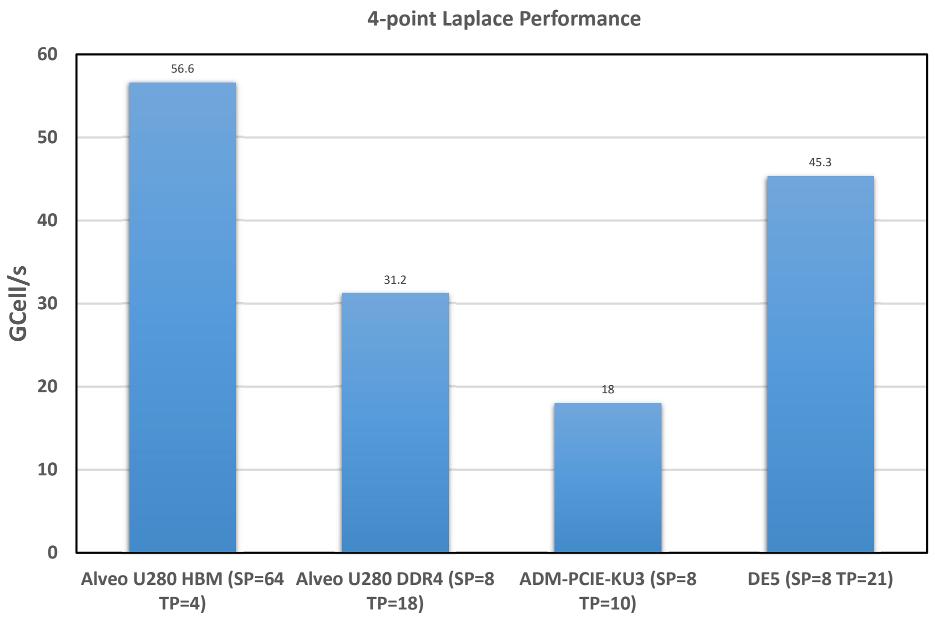


Figure 5.15: The 4-point Laplace Equation benchmark performance

Figure 5.15 presents the evaluation results of the 4-point Laplace Equation. The target stencil array is set to (16,384 × 16,384) in this benchmark. The performance results of the DE5 and ADM-PCIE-KU3 FPGA boards are from the previous studies [50] and [55], respectively. Due to the constraints of the external memory bandwidth, they can only use 8 as the maximum SP value. However, unlike the Sobel 2D stencil application, they can still increase the stencil performance by using the temporal-based parallelization. Compared with their studies, the SP value of the stencil computation architecture can be 64 with the HBM as external memory and 8 with the DDR as external memory.

Figure 5.16 shows the power estimation report which is generated by the implementation tool. I set the default junction temperature as 37.9 °C. The report shows the total required power of FPGA chip is 30.83 W. I list the top 5 hardware elements in term of the power consumption. The most power is consumed by the HBM, i.e., 34%. The logic part, e.g., LUTs, BRAMs consumes 23%. The other elements including GTY, clock, and signal cost about 40%.

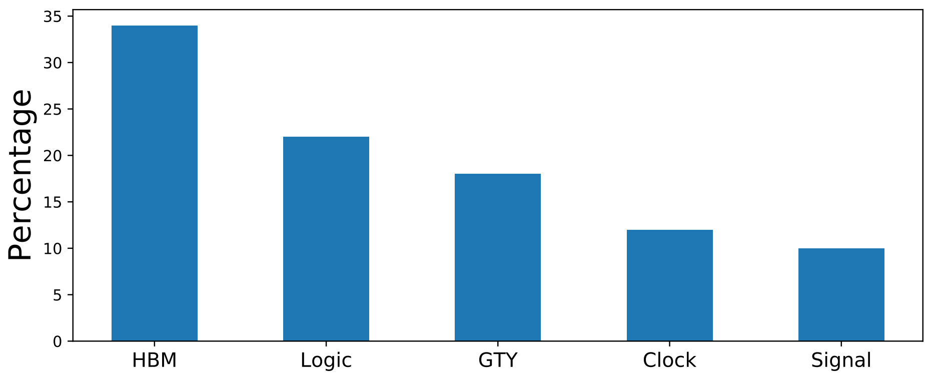


Figure 5.16: Power report of the 4-point Laplace Equation benchmark

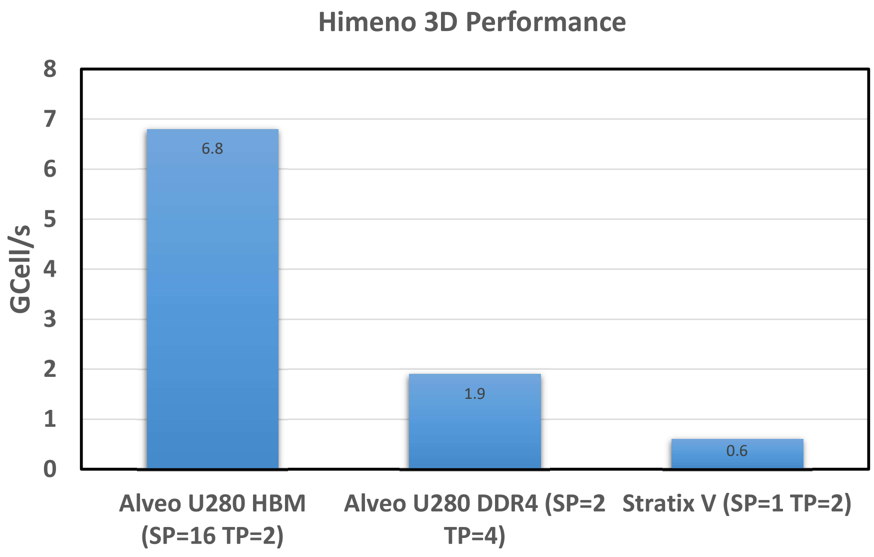


Figure 5.17: The Himeno benchmark performance

Figure 5.17 presents the performance results for the 3D Himeno benchmark. The target stencil array is $(256 \times 256 \times 512)$. For the 3D stencil arrays, the custom buffer needs to store more stencil cells than the 2D arrays. For example, in 2D the custom buffer size is mainly decided by 1 dimension length of the array. For 3D array, the custom buffer needs to store the 2D planes which are decided by the product of 2 dimension length of the array. As a result, the memory resource consumption of the stencil architecture often becomes the bottleneck of performance. As I stated in the performance model, using large TP value may exhaust the memory resources before the peak performance is reached. In this benchmark, the largest allowed TP value is 4. However, with the aid of HBM, I can use the SP value 16 to employ the spatial-based parallelization to increase the performance. The results show that the computation architecture can achieve almost 4 times higher performance in the HBM-based design compared to the DDR-based.

5.5 Summary

In this chapter, I introduce the stencil architecture design on a HBM connected FPGA. By using the HBM as external memory, the limitations of the spatial-based parallelization is removed and the design space is expanded. As a result, I can scale the stencil architecture performance by considering the spatial-based parallelization and temporal-based parallelization equally. I also extend the custom buffer design into multiple dimension spaces of the target stencil array.

I evaluate the proposed stencil architecture on the Xilinx Alveo U280 FPGA board with 3 typical benchmarks. In the Sobel 2D filter, I can obtain the 128.4 GCell/s with the 8-bit integer data type pictures. For the 4-point Laplace Equation, I achieve the result of 56.6 GCell/s. At last, in the 3D Himeno benchmark, the stencil architecture performance can reach to 6.8 GCell/s.

Chapter 6

Conclusion and Future Work

In this chapter, I conclude the contributions of this thesis paper and show future work of out study.

6.1 Conclusions

In this thesis paper, my main contribution is to propose the stencil computation architecture based on FPGAs by using the HLS developing method. The HLS compiler uses the programming language like C/C++ or OpenCL to describe hardware implementation. It abstracts the execution procedure of the design into loops, functions, and the on-chip memory system is built by the combinations of variables and arrays. Although HLS has got a lot of great achievements in the past decade, it is still not smart enough to take a simple high-level description of an FPGA design and transfer it into an efficient hardware implementation.

As I stated in this thesis, to achieve high performance for stencil applications on the FPGAs with the HLS method, the users need to let the HLS tools automatically build the high efficiency pipeline system of the stencil computations, I have specifically described a custom buffer structure to help the HLS avoid the data dependency and generate stalls in the pipeline system. Compared with the RTL-based FPGA design, although the users of HLS tools can easily use the pragma, e.g., *loop unroll* in the loop nests to exploit the SIMD-like parallelism and do not need to explicitly alter the control signals of the loop structure, the HLS compiler may not prepare an optimal memory system for the computing logic. The proposed custom buffer design also can be extended to bind the parallelism and execution together by using the optimal hardware resources. To efficiently use the external memory bandwidth of the target FPGA board, I have shown the optimization techniques of how to describe a proper memory interface contention with the HLS.

In the first part of this thesis, I use 2 specific stencil applications of CFD, i.e., LBM and LGCA simulations to explain the proposed architecture design. I propose a

FIFO-based custom buffer design to exploit the data reusability both in the spatial and temporal domain. I explicitly describes the data movements between the registers and FIFOs to avoid the dependence on the special IP core. Moreover, in LBM simulation, I propose an extended custom buffer design that can provide sufficient data bandwidth to multiple CUs in the same time step without duplicating the same data in the custom buffer. I evaluate the simulation results on the Xilinx VCU1525 FPGA board. The results show that by using both spatial-based and temporal-based parallelization, the proposed computation architecture can achieve competitive performance compared to the other computing devices, e.g. CPUs and GPUs.

In the second part of this thesis, based on the work in the first part, I present a stencil computation architecture design for FPGAs used the HBM as the external memory. Due to the aid of HBM, the memory bandwidth of FPGAs increases significantly, which opens the new optimization opportunities. To fully explore the design space, I extend the custom buffer design to multiple dimensions of the target stencil array to exploit the optimizations of computation results sharing inside the certain stencil applications. I evaluate the computation architecture with the 3 benchmarks, i.e., Sobel 2D filter, 4-point Laplace Equation, and 3D Himeno on the target Xilinx Alveo U280 FPGA board. The performance results show that for the stencil applications with limited temporal parallelism, the high external memory bandwidth can help FPGA to achieve one order of magnitude performance gains than the traditional FPGAs. And the resource consumption shows that by extending the design space of custom buffer the optimization of sharing the computation result can be utilized.

Compared with the previous studies, I have solved 3 existed challenges. First, the Shift register IP core dependence. I use the FIFOs and registers to explicitly describe the custom buffer behaviour instead of using the shift register-based behaviour description which needs the support of the specific IP core of Intel. The stencil applications in this thesis paper, e.g., LGCA, LBM, Himeno have used the proposed custom buffer to avoid the dependency of the data accesses in multiple time dimensions. Second, the previous work uses duplicated custom buffer to feed data to the computing logic to increase parallelism of the stencil computations. The proposed custom buffer design in this thesis costs optimal hardware resources to provide enough data bandwidth to the computing logic. For example, in the case of LBM, the memory resource consumption of the SP = 2, is almost half compared to the SP = 1. At last, the share of the calculated result among the multiple stencil operations is also enable in the proposed stencil computation architecture. Instead of increase the spatial-based pluralization with limited design space, I extend the design space to the multiple space dimensions. In the case of the stencil applications, e.g., Sobel 2D, and Laplace equation, the resource consumption of the computing logic is decreased along with the sharing of the calculated results among the multiple stencil operations.

6.2 Future Work

In future study, I agree that the HLS design portability is an interesting topic to discuss. Through the HLS, I can abstract the architecture design description with high-level languages and do not need to care too much about the details in dividing the pipeline stages, synchronizing functions by certain clock cycles, or FSM control signals. Therefore, the potential portability in FPGA design is raising. However, compared to high-level language design based on fixed architecture e.g., CPUs, the HLS design with FPGAs is still hard to move to another HLS platform.

Currently, the main object of this paper is not to solve the portability issue and present general guidelines for improving portability among different HLS tools. However, I want to discuss the HLS potability problem and provide the hints to help developers realize which parts of the code might compromise the portability of the design in future work. In terms of portability, I suggest that developers should carefully utilize HLS compiler-relevant techniques. For the developers who do not have a strong background with FPGA, these techniques may prevent them from identifying the performance bottleneck when porting the design to other platforms. To achieve the similar portable level as the high-level language tools on CPUs, the HLS tools on FPGAs still have a long way to go. It needs both academia and industry to continuously contribute to this area, such as standard IP core behaviours, unified compiler pragmas and memory interface protocols. In addition, due to the feature of the high-level languages, using the middle representation of these languages to build a tool that can automatically generate the HLS-based stencil design with different design parameters is also interesting. The tool can help users to preform the parameter tuning and search the best design parameter for the target stencil application with the specific FPGA board.

Another thing draw my attention is the operating frequency of the FPGA design. To obtain better performance, the operating frequency of the stencil computation architecture on FPGA is also can be increased. However, the architecture design with high resource utilization often puts heavy pressure on the routing resource, which prevents us to use larger value of the operating frequency. At the current stage, the approaches and guidelines to scale up design frequency with the HLS development method are still unclear. Since by using the HLS, the detail control of hardware design is lost. For example, it is very hard to identify the critical path in the HLS design code and use the corresponding techniques to optimize it. The pipeline stages are also decided by the HLS compiler, the users also cannot control the pipeline system specifically. In addition, the current HLS tools can not predict the accurate frequency with the consideration of routing resources, unless the placement and routing procedure are actually finished, which will consume a lot of time. I will try to address these problems in future research.

Bibliography

[1] Moore, Gordon E. ”Cramming more components onto integrated circuits.” Proceedings of the IEEE 86.1 (1998): 82-85.

[2] Dennard, Robert H., et al. ”Design of ion-implanted MOSFET’s with very small physical dimensions.” IEEE Journal of Solid-State Circuits 9.5 (1974): 256-268.

[3] Feichtinger, Christian, et al. ”Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU–GPU clusters.” Parallel Computing 46 (2015): 1-13.

[4] Johnson, Mitchel, Daniel P. Playne, and Kenneth A. Hawick. “Data-Parallelism and GPUs for Lattice Gas Fluid Simulations,” Proc. International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 210-216, 2010.

[5] Phillips, Everett H., and Massimiliano Fatica. ”Implementing the Himeno benchmark with CUDA on GPU clusters.” 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE, 2010.

[6] Vestias, Mario, and Horácio Neto. ”Trends of CPU, GPU and FPGA for high-performance computing.” 2014 24th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2014.

[7] Putnam, Andrew, et al. ”A reconfigurable fabric for accelerating large-scale datacenter services.” 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). IEEE, 2014.

[8] Vanderbauwhede, Wim, and Khaled Benkrid, eds. High-performance computing using FPGAs. Vol. 3. New York, NY, USA:: Springer, 2013.

[9] Jones, David H., et al. ”GPU versus FPGA for high productivity computing.” 2010 International Conference on Field Programmable Logic and Applications. IEEE, 2010.

[10] Asano, Shuichi, Tsutomu Maruyama, and Yoshiki Yamaguchi. ”Performance comparison of FPGA, GPU and CPU in image processing.” 2009 international conference on field programmable logic and applications. IEEE, 2009.

[11] Project Catapult. Available online: `https://www.microsoft.com/en-us/research/project/project-catapult/`

[12] Arcas-Abella, Oriol, et al. "An empirical evaluation of high-level synthesis languages and tools for database acceleration." 2014 24th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2014.

[13] Vivado High-Level Synthesis. Available online: `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`

[14] Intel High Level Synthesis Compiler. Available online: `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`

[15] Canis, Andrew, et al. "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems." ACM Transactions on Embedded Computing Systems (TECS) 13.2 (2013): 1-27.

[16] Datta, Kaushik, et al. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures." SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE, 2008.

[17] Maruyama, Naoya, and Takayuki Aoki. "Optimizing stencil computations for NVIDIA Kepler GPUs." Proceedings of the 1st international workshop on high-performance stencil computations, Vienna. 2014.

[18] Trimberger, Stephen M. Steve. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation." IEEE Solid-State Circuits Magazine 10.2 (2018): 16-29.

[19] UltraScale Architecture DSP Slice User Guide. Available online: `https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf`

[20] 7 Series FPGAs Memory Resources User Guide. Available online: `https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`

[21] Kilts, Steve. Advanced FPGA design: architecture, implementation, and optimization. John Wiley & Sons, 2007.

[22] Keating, Michael, and Pierre Bricaud. Reuse Methodology Manual for System-on-a-Chip Designs: For System-on-a-chip Designs. Springer Science & Business Media, 2002.

[23] Donzellini, Giuliano, et al. ”Introduction to FPGA and HDL Design.” Introduction to Digital Systems Design. Springer, Cham, 2019. 465-517.

[24] Cong, Jason, et al. ”High-level synthesis for FPGAs: From prototyping to deployment.” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30.4 (2011): 473-491.

[25] Introduction to FPGA Design with Vivado High-Level Synthesis. Available online: `https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf`

[26] Kuchcinski, Krzysztof. ”Constraints-driven scheduling and resource assignment.” ACM Transactions on Design Automation of Electronic Systems (TODAES) 8.3 (2003): 355-383.

[27] Marwedel, Peter. ”The MIMOLA design system: Tools for the design of digital processors.” 21st Design Automation Conference Proceedings. IEEE, 1984.

[28] Granacki, John, David Knapp, and Alice Parker. ”The ADAM advanced design automation system: overview, planner and natural language interface.” 22nd ACM/IEEE Design Automation Conference. IEEE, 1985.

[29] Paulin, Pierre G., John P. Knight, and Emil F. Girczyc. ”HAL: a multi-paradigm approach to automatic data path synthesis.” 23rd ACM/IEEE Design Automation Conference. IEEE, 1986.

[30] Knapp, David W. Behavioral synthesis: digital system design using the synopsys behavioral compiler. Prentice-Hall, Inc., 1996.

[31] Fredriksson, M. ”Visual Architect Overview Architecture and Algorithms.” Cadence Technical Conference. 1997.

[32] Elliott, John P. Understanding behavioral synthesis: a practical guide to high-level design. Springer Science & Business Media, 1999.

[33] Intel SDK For OpenCL Applications. Available online: `https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html`

[34] Pilato, Christian, and Fabrizio Ferrandi. ”Bambu: A modular framework for the high level synthesis of memory-intensive applications.” 2013 23rd International Conference on Field programmable Logic and Applications. IEEE, 2013.

[35] Lattner, Chris, and Vikram Adve. ”LLVM: A compilation framework for lifelong program analysis & transformation.” International Symposium on Code Generation and Optimization, 2004. CGO 2004.. IEEE, 2004.

[36] Datta, Kaushik, et al. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures." SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE, 2008.

[37] Huynh, H. T., Zhi J. Wang, and Peter E. Vincent. "High-order methods for computational fluid dynamics: A brief review of compact differential formulations on unstructured grids." Computers & fluids 98 (2014): 209-220.

[38] Taflove, Allen, and Susan C. Hagness. Computational electrodynamics: the finite-difference time-domain method. Artech house, 2005.

[39] Smith, Gordon D., Gordon D. Smith, and Gordon Dennis Smith Smith. Numerical solution of partial differential equations: finite difference methods. Oxford university press, 1985.

[40] Hegarty, James, et al. "Darkroom: compiling high-level image processing code into hardware pipelines." ACM Trans. Graph. 33.4 (2014): 144-1.

[41] Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." Communications of the ACM 52.4 (2009): 65-76.

[42] Datta, Kaushik, et al. "Optimization and performance modeling of stencil computations on modern microprocessors." SIAM review 51.1 (2009): 129-159.

[43] Nguyen, Anthony, et al. "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs." SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2010.

[44] Pohl, Thomas, et al. "Optimization and profiling of the cache performance of parallel lattice Boltzmann codes." Parallel Processing Letters 13.04 (2003): 549-560.

[45] Williams, Samuel, et al. "Scientific computing kernels on the cell processor." International Journal of Parallel Programming 35.3 (2007): 263-298.

[46] Wellein, Gerhard, et al. "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization." 2009 33rd Annual IEEE International Computer Software and Applications Conference. Vol. 1. IEEE, 2009.

[47] Cong, Jason, et al. "Automatic memory partitioning and scheduling for throughput and power optimization." ACM Transactions on Design Automation of Electronic Systems (TODAES) 16.2 (2011): 1-25.

[48] Wang, Yuxin, et al. "Memory partitioning for multidimensional arrays in high-level synthesis." Proceedings of the 50th Annual Design Automation Conference. 2013.

[49] Zohouri, Hamid Reza, Artur Podobas, and Satoshi Matsuoka. "Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL." Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2018.

[50] Waidyasooriya, Hasitha Muthumala, et al. "OpenCL-based FPGA-platform for stencil computation and its optimization methodology." IEEE Transactions on Parallel and Distributed Systems 28.5 (2016): 1390-1402.

[51] Nacci, Alessandro Antonio, et al. "A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices." 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, 2013.

[52] Takei, Yasuhiro, et al. "FPGA-oriented design of an FDTD accelerator based on overlapped tiling." Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.

[53] Wang, Shuo, and Yun Liang. "A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model." Proceedings of the 54th Annual Design Automation Conference 2017.

[54] de Fine Licht, Johannes, Michaela Blott, and Torsten Hoefler. "Designing scalable FPGA architectures using high-level synthesis." Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2018.

[55] Chi, Yuze, et al. "SODA: stencil with optimized dataflow architecture." 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2018.

[56] Intel RAM-Based Shift Register IP core. Available online: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_shift_register_rambased.pdf`

[57] Xilinx RAM-based Shift Register. Available online: `https://www.xilinx.com/products/intellectual-property/ram-based_shift_register.html`

[58] Wolf-Gladrow, Dieter A. Lattice-gas cellular automata and lattice Boltzmann models: an introduction. Springer, 2004.

[59] Zheng, Zhong, et al. "A dual-scale lattice gas automata model for gas–solid two-phase flow in bubbling fluidized beds." Computers & Mathematics with Applications 61.12 (2011): 3593-3605.

[60] Mente, Carsten, Anja Voss-Böhme, and Andreas Deutsch. "Analysis of individual cell trajectories in lattice-gas cellular automaton models for migrating cell populations." Bulletin of mathematical biology 77.4 (2015): 660-697.

[61] Frisch, Uriel, Brosl Hasslacher, and Yves Pomeau. "Lattice-gas automata for the Navier-Stokes equation." Physical review letters 56.14 (1986): 1505.

[62] Johnson, Mitchel, Daniel P. Playne, and Kenneth A. Hawick. "Data-Parallelism and GPUs for Lattice Gas Fluid Simulations." PDPTA. 2010.

[63] Chen, Shiyi, and Gary D. Doolen. "Lattice Boltzmann method for fluid flows." Annual review of fluid mechanics 30.1 (1998): 329-364.

[64] Wittmann, Markus, et al. "Comparison of different propagation steps for lattice Boltzmann methods." Computers & Mathematics with Applications 65.6 (2013): 924-935.

[65] Tomczak, Tadeusz, and Roman G. Szafran. "Sparse geometries handling in lattice Boltzmann method implementation for graphic processors." IEEE Transactions on Parallel and Distributed Systems 29.8 (2018): 1865-1878.

[66] Kuznik, Frédéric, et al. "LBM based flow simulation using GPU computing processor." Computers & Mathematics with Applications 59.7 (2010): 2380-2392.

[67] Maximizing Memory Bandwidth with Vitis and Xilinx UltraScale+ HBM Devices. Available online: `https://developer.xilinx.com/en/articles/maximizing-memory-bandwidth-with-vitis-and-xilinx-ultrascale-hbm-devices.html`

Acknowledgements

Firstly, I would like to show my sincerely gratitude to my research supervisor, Professor Yoshiki Yamaguchi who continually supports me with his guidance, experience, and enthusiasm. The most important is he offers me the opportunity to join his lab in 2017 where I can pursue my dream of starting my academic career. In addition to the research, he also cares about my daily life, especially for the health situation of my back hurts.

Besides of the Prof. Yamaguchi, I would like to thank my colleges in the FPGA lab for their friendship and communications. They will be a great memory in my life. I also would like to thanks Prof. Boku, the head of CCS, the MEXT, and JASSO to support me with the financial aids. These aids really help me to dedicate myself to my studies.

Finally, I always thank my family for their strong efforts to support me during my studies in Japan. My wife and my newborn son, they sacrifice a lot for me, especially in this special time. They are my strong motivation to finish my Ph.D.

List of Publications

Reviewed Journal

- Du, Changdao; Yamaguchi, Yoshiki; High-Level Synthesis Design for Stencil Computations on FPGA with High Bandwidth Memory, Electronics 2020, Volume 9 Issue 8, pp.1-19

Reviewed Conference

- Du, Changdao; Firmansyah, Iman; Yamaguchi, Yoshiki; High-Performance Computation of LGCA Fluid Dynamics on an FPGA-based Platform, Proceeding of International Conference on Computer and Communication Systems, pp520-525, 2020-05-15–2020-05-18
- Du, Changdao; Firmansyah, Iman; Yamaguchi, Yoshiki; FPGA-Based Computational Fluid Dynamics Simulation Architecture via High-Level Synthesis Design Method, Proceedings of Applied Reconfigurable Computing, pp.232-246, 2020-04-01–2020-04-03
- Du, Changdao; Yamaguchi, Yoshiki; A High-Level Synthesis Design for a Scalable Hydrodynamic Simulation on OpenCL FPGA Platform, Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, pp.1-4, 2019-06-06–2019-06-07

Reviewed Conference (Coauthor)

- Firmansyah, Iman; Changdao, Du; Fujita, Norihisa; Yamaguchi, Yoshiki; Boku, Taisuke; FPGA-based Implementation of Memory-Intensive Application using OpenCL, Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, pp.1-4, 2019-06-06–2019-06-07