

**FPGA ACCELERATION OF
STENCIL-BASED COMPUTATIONS**

BY

ALHUSSAIN IBRAHIM

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

MAY/ 2023

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **ALHUSSAIN IBRAHIM** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING**.

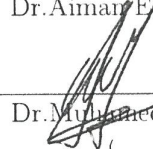
Thesis Committee



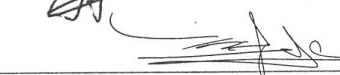
Dr. Muhammad Elrabaa (Adviser)



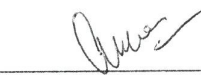
Dr. Aiman El-Maleh (Co-Adviser)



Dr. Mohamed Mudawar (Member)



Dr. Abdulaziz Tabbakh (Member)



Dr. Ayaz Khan (Member)



Dr. Aiman El-Maleh
Department Chairman



Dr. Suliman Al-Homidan
Dean of Graduate Studies

Date



©Alhussain Ibrahim
2023

Dedication

"To my guiding star, my father. Your strength shapes my foundation, your wisdom lights my path. In your quiet sacrifices, I find the loudest love."

"To the heartbeat of my existence, my mother. Your love, a silent force, molds my essence. In your embrace, I discover the purest form of home."

"To my brothers and sister, companions in adventure and allies in adversity. Through shared laughter and unspoken bonds, you've enriched my world. In the book of family, your chapters are my favorite stories of camaraderie and support."

ACKNOWLEDGMENTS

All praise and thanks be to Almighty Allah, the most Gracious, the most Merciful.

Peace and mercy be upon His Prophet and his progeny.

I extend my heartfelt gratitude to my esteemed advisor, Prof. Muhammed El-Rabaa, whose guidance and unwavering support have been the cornerstones of this academic journey.

I would also like to express my sincere appreciation to my co-advisor, Prof. Aiman El-Maleh, for their invaluable input and expertise. Your collaborative approach has enriched the depth and quality of my research.

A special thanks to the members of my thesis committee, Dr. Muhamed Mudawar, Dr. Abdulaziz Tabbakh and Dr. Ayaz Khan, for their time, expertise, and thoughtful critique. Your diverse perspectives have been instrumental in refining the scope and methodology of this work.

I would also like to express my sincere gratitude to Mr. Saleh Alsaleh for his invaluable help to the completion of this thesis.

Lastly, I extend my deepest gratitude to my family, whose unwavering support has been the bedrock of my academic journey.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	vi
LIST OF TABLES	ix
LIST OF FIGURES	xi
ABSTRACT (ENGLISH)	xiv
ABSTRACT (ARABIC)	xvi
CHAPTER 1 INTRODUCTION AND BACKGROUND	1
1.1 Stencil Computations	3
1.2 OpenCL framework	5
CHAPTER 2 LITERATURE REVIEW	8
2.1 Data Locality and Computation Parallelism	9
2.2 Data buffering	12
2.3 Floating Point Numbers	13
2.4 Stencil Implementations on FPGA	14
CHAPTER 3 PROPOSED FRAMEWORK	16
3.1 Stencil Kernel	19
3.1.1 SODA Framework	19
3.1.2 Coalesce Memory Transfers and Kernel Execution	21
3.2 Host Application	22
3.2.1 Overlapping tiles	25

3.2.2 Extending the Framework to Multiple FPGA Devices	27
CHAPTER 4 EXPERIMENTAL RESULTS	30
4.1 Experimental Setup	30
4.2 Stencil Kernel Evaluation	31
4.3 SODA Modification to Reduce FPGA Resource Utilization	39
4.4 Evaluating the Effect of Tile Size on Tiling Stage Performance	52
4.5 Evaluating the Effect of Tile Size on Sending/Receiving To/From FPGA's Global Memory	53
4.6 Evaluating the Overall Performance of the Framework	54
4.7 Comparing Framework Performance to CPU-Based and GPU-Based Implementations	56
4.8 Multi-FPGA Framework Simulation	59
CHAPTER 5 CONCLUSIONS AND FUTURE WORK	61
REFERENCES	63
VITAE	68

LIST OF TABLES

4.1	Specification of the host machine.	30
4.2	Throughput in (GB/s) versus Unrolling factors (UF) and number of fused time steps (Ts) for three tile sizes.	32
4.3	On-FPGA Block RAMs (M20K Blocks) Usage.	33
4.4	Adaptive Logic Module (ALM) usage in thousands.	34
4.5	Digital Signal Processing (DSP) blocks Usage.	34
4.6	Memory Logic Array Block (MLAB) usage.	35
4.7	Registers usage in thousands.	35
4.8	Obtained on-FPGA frequencies versus Unrolling factors (UF) and num- ber of fused time steps (Ts) for three grid sizes.	36
4.9	Average power consumption in Watts versus Unrolling factors (UF) and number of fused time steps (Ts) for three tile sizes.	37
4.10	Power dissipation components breakdown for a grid size of 128x128x1,024, UF=4, Ts=5.	37
4.11	FPGA resource utilization, attained frequency, power consumption, and throughputs obtained for the different kernel clustering options for a slice size of 128×128 , UF=1, and Ts=1. Maximum Power Efficiency = 16.5 W/GB/s (for the single kernel).	50
4.12	FPGA resource utilization, attained frequency, power consumption, and throughputs obtained using the single kernel for a slice size of 128×128 , UF=1, and Ts=5. Maximum Power Efficiency = 3.83 W/G- B/s (for the Full kernel clustering).	50

4.13 FPGA resource utilization, attained frequency, power consumption, and throughputs obtained using the single kernel for a slice size of 128×128 , $UF=4$, and $Ts=5$. Maximum Power Efficiency = 0.99 W/GB/s (for the single kernel).	50
4.14 FPGA resource utilization, attained frequency, power consumption, and throughputs obtained for the different kernel clustering options for a slice size of 256×256 , $UF=1$, and $Ts=1$. Maximum Power Efficiency = 17.65 W/GB/s (for the single kernel).	51
4.15 FPGA resource utilization, attained frequency, power consumption, and throughputs obtained using the single kernel for a slice size of 256×256 , $UF=1$, and $Ts=5$. Maximum Power Efficiency = 5.06 W/GB/s (for the Full kernel clustering).	51
4.16 FPGA resource utilization, attained frequency, power consumption, and throughputs obtained using the single kernel for a slice size of 256×256 , $UF=4$, and $Ts=5$. Maximum Power Efficiency = 1.32 W/GB/s (for both, the single and Full kernel clustering).	51
4.17 Power dissipation components breakdown of single kernel version for a grid size of $128 \times 128 \times 1,024$, $UF=4$, $Ts=5$.	52
4.18 Tile dimensions used to evaluate tiling stage performance.	52
4.19 Tiling throughput (GB/s) as a function of fast x (z-axis) and intermediate (y-axis) dimensions.	53
4.20 Total latency in seconds of computing the whole 3-D grid (from the start of tiling the first tile till the untiling of the last tile) versus tile size and Ts .	56
4.21 Minimum latencies in milliseconds of the asynchronous pipeline stages for three sizes as a function of Ts .	57
4.22 Total latency in seconds of computing the whole 3-D grid versus tile size and Ts for one device and two devices.	60
4.23 Number of tiles per device versus tile size and Ts for one device and two devices.	60

LIST OF FIGURES

1.1	Different shapes of stencil.	4
1.2	Mapping of a 7-point 3D stencil's data at point p to RAM memory.	4
1.3	Execution model of OpenCL on FPGA.	6
1.4	Memory model of OpenCL.	7
2.1	An example of combined iterations in 5-points stencil.	10
2.2	Multiple stencil iteration processing using pipeline of processing elements.	11
2.3	Sliding window buffer.	12
3.1	Framework stages.	17
3.2	Execution of the framework pipelined stages.	18
3.3	25-point stencil shape.	19
3.4	Design of SODA reuse buffer.	21
3.5	Overview of SODA microarchitecture.	22
3.6	Illustration of the double buffering technique on the FPGA.	23
3.7	The proposed framework.	25
3.8	Overlapped tiles.	27
3.9	Execution example of multiple buffering technique in case of: A. producing data B. Consuming data.	29
3.10	Boundary exchange between the two sub-grids: boundary tiles are shown in yellow colour, while the exchanged boundary points are shaded in brown.	29
4.1	Code snippet showing the two non-autorun kernels for reading and writing from/to the global memory.	39

4.2	Code snippet of an autorun kernel of a forwarding module (FIFO).	40
4.3	Code snippet of an autorun kernel for a compute module (PE).	41
4.4	Code snippet of combined forwarding and compute modules in SODA using (coarse) clustering level.	42
4.5	Code snippet of combined stages in SODA using the (full) clustering level.	43
4.6	The code for the shift register replacing the forwarding module in the none clustering option.	44
4.7	Code snippet of a kernel with combined FIFO modules.	45
4.8	Code snippet of a kernel with combined FIFO and compute modules.	46
4.9	Code snippet of the developed single kernel version.	47
4.10	The sending and receiving functions utilizing OpenCL APIs.	54
4.11	Throughput vs. block size.	55
4.12	Comparison between the framework and both CPU work on [1] and [2].	57
4.13	Comparison between the framework and GPU-based implementation in [3].	58

THESIS ABSTRACT

NAME: Alhussain Ibrahim
TITLE OF STUDY: FPGA Acceleration of Stencil-Based Computations
MAJOR FIELD: Computer Engineering
DATE OF DEGREE: May 2023

Stencil computations are widely used in many High Performance Computations (HPC) applications. This type of computations show a great potential for parallelism making them especially appealing in the context of parallel computing. However, they suffer from large memory footprint, low arithmetic intensity, and irregular data access patterns. Many attempts were done in the literature to overcome these challenges using different platforms. Among these platforms, the FPGAs which are known for their hardware configurability and power efficiency. FPGAs, however, suffers from limited resources and memory that put restriction on the size of stencil data that could be processed. A tiling technique should be incorporated to alleviate this restriction. In this thesis, a pipelined FPGA-CPU stencil framework is introduced. This framework aims to overlap the tiling process in CPU with memory transfers and stencil computation in the FPGA. Experimental results show that this framework achieved a throughput

of 36 GB/s using Intel’s Stratix 10 SX Development Board. The FPGA was fabricated using a relatively 14 nm technology. This performance is equivalent to that of ~43 Haswell (fabricated in a 22 nm technology) cores or ~13 Milan-x (fabricated in a 7 nm technology) cores. In terms of power efficiency, this framework achieved 1.32 W/GB/s which represents 70% of the power efficiency of NVIDIA A100 GPU (with 6,912 CUDA cores fabricated in a 7nm technology) implementation. Additionally, it was demonstrated that the developed framework supports the utilization of two FPGA devices in parallel with 60% reduction in execution latency.

ملخص الرسالة

الاسم: الحسين ابراهيم

عنوان الدراسة: تسريع العمليات الحسابية المقولبة على أجهزة ال(FPGA)

التخصص: هندسة الحاسب الآلي

تاريخ الدرجة العلمية: شوال 1444هـ

العمليات الحسابية المقولبة (Stencil-based computations) تستخدم بشكل واسع في مجال الحوسبة فائقة الأداء. هذا النوع من العمليات يمكنها الاستفادة من الحوسبة المتوازية، لكن وفي المقابل العمليات الحسابية المقولبة لديها تأثير سيء على الذاكرة بسبب كثرة استخدامها للذاكرة مقارنة بعدد العمليات الحسابية التي تقوم بها. الكثير من الباحثين حاولوا التغلب على الصعوبات التي يواجهونها في هذه العمليات على منصات مختلفة. من بين هذه المنصات، تبرز أجهزة ال(FPGA) لمرونة عتادها وكفاءة استخدامها للطاقة لكنها تعاني من شح مواردها العتادية وضعف ذواكرها مما يضع قيوداً على حجم البيانات التي يمكن معالجتها عليها. هذه المشكلة يمكن حلها بتقسيم البيانات إلى أجزاء أصغر يمكن معالجتها على ال(FPGA). هذه الرسالة تقدم إطاراً (framework) لمعالجة العمليات الحسابية المقولبة. هذا الإطار يستخدم أنابيب التجهة (pipeline) ليدخل عمليات نقل البيانات مع عمليات معالجتها. استطاع هذا الإطار الوصول إلى سرعة 36GB/s باستخدام (Intel's Stratix 10 SX Development Board). هذا الجهاز تمت صناعته باستخدام تقنية 7 نانومتر. الأداء الذي قدمه هذا الإطار يعادل تقريبا 43 نواة من معالج (Haswell) الذي تمت صناعته بتقنية 22 نانومتر ويعادل تقريبا 13 نواة من معالج (Milan-x) الذي تمت صناعته بتقنية 7 نانومتر. وقد حقق الإطار أيضا كفاءة طاقة تعادل 1.32 W/GB/s والتي تمثل 70% من كفاءة الطاقة لمعالج الرسومات (NVIDIA A100) والذي يحتوي على 6912 نواة وتمت صناعته بتقنية 7 نانومتر.

CHAPTER 1

INTRODUCTION AND BACKGROUND

The ever-increasing demand for high computational power and the slowing pace of technology scaling have changed the direction of computations from single-processor architectures to multicore architectures. However, this has been accompanied by new challenges such as the amount of parallelizable work in a program, load balancing, and synchronization between cores. To overcome some of these challenges, specialized architectures, such as GPUs, and reconfigurable hardware platforms, such as FPGAs, were used as accelerators. These accelerators have shown great results. To improve the computational performance even further, heterogeneous systems composed of different platforms were used. Such systems have further increased the challenges facing the application developers. Nevertheless, the emergence of heterogeneous computing frameworks such as OpenCL has eased the development in different platforms using a single language supported by a common framework for developing distributed

applications.

Recently, due to their hardware configurability and inherent parallel structure, FPGAs have been increasingly used as accelerators for high-performance computing (HPC applications. They were shown to provide high computational throughputs at lower energy consumption levels compared to their GPU and multi-core CPU counterparts.

Many HPC applications utilize stencil-based computations (e.g. Finite Difference) as the main computation kernel. One or more stencils (of various degrees) are applied to one/two/three dimensional data arrays over many iterations in time. Such applications are ideal for acceleration due to the inherent data parallelism. One of such applications is Reverse Time Migration (RTM) of Seismic waves, a technique used to explore sub-surface earth layers in order to identify potential Oil or Gas reserves. Stencil computations, however, suffers from large memory footprint, low arithmetic intensity, and irregular data access patterns.

In this thesis, an efficient framework for accelerating the computation of high-order stencils on large 3-D grids on FPGAs is explored. A complete framework for accelerating such a computation based on a fully asynchronous pipeline has been developed, implemented, and evaluated.

The rest of this thesis is organized as follow. background information on Stencil computations on FPGAs and the OpenCL framework will be introduced through the rest of this chapter. Chapter 2 presents a comprehensive literature review of the state of the art in Stencils implementations on FPGA. Chapter 3 provide a detailed description of the proposed framework. Experimental results for evaluating the pro-

posed framework in terms of performance, resource usage, and power efficiency will be presented in chapter 4. Finally, Chapter 5 provides the conclusions and suggested future work.

1.1 Stencil Computations

Stencil computations are iterative grid-based operations where each cell on a grid on a specific iteration is updated based on the values of its neighbours from the previous iteration. The applications of Stencil computations are not limited to a specific field. In fact they are used in different domains including, but not limited to, image processing, simulation of fluid dynamics, and solving differential equations. The positions and the number of neighbouring elements needed for each computation varies depending on the Stencil's application. Figure 1.1 shows different structures of 2-D and 3-D stencils. Higher order stencils use more neighbouring points (e.g. 4 points in each direction). Since multidimensional data, usually specified as multidimensional arrays of floating point numbers, are linearized memory (stored as 1-D arrays), the neighbouring points in a stencil are not necessarily adjacent in memory. In fact, only neighbours along one axis are adjacent. E.g. Intel compilers stores multidimensional arrays using the last dimension first, then the next one, then the next one and so on till the first dimension. So, for a 3-D array, data are stored Z-first, then Y, then X (hence X is called the slow dimension). Figure 1.2 illustrates how Stencil's neighbouring elements are stored in memory.

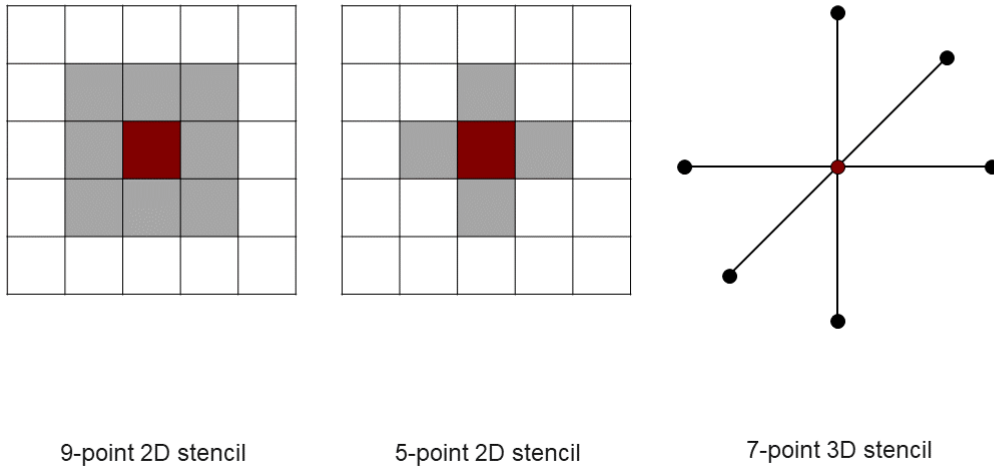


Figure 1.1: Different shapes of stencil.

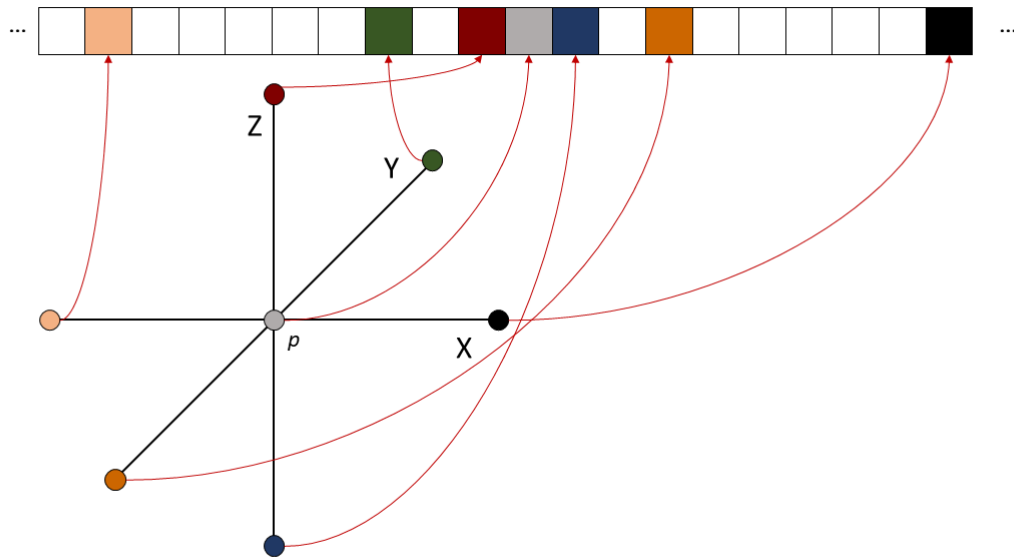


Figure 1.2: Mapping of a 7-point 3D stencil's data at point p to RAM memory.

1.2 OpenCL framework

Open Computing Language (OpenCL) is a framework for heterogeneous computing managed by the non-profit organization Khronos Group [4]. Codes written in OpenCL can be executed across multiple platforms such as central processing units (CPUs), graphics processing units (GPUs), and field-programmable gate arrays (FPGAs) [4].

The OpenCL platform model consists of one or more OpenCL devices connected to a host. Each of these devices has one or more compute units and each compute unit is composed of one or more processing elements (PE). A system could have multiple platforms. However, different platforms usually do not interact with each other. Each platform corresponds to an SDK (software development kit) provided by vendors.

In the execution model, Opencl specification defines two execution units: host application and kernel. Host applications are usually written in a general programming language (such as C or C++) and compiled to be executed on the CPU host machine. In host application, a context is created and associated with one OpenCL device or more. This context provides an environment to manage devices, memories and control their interactions. For each device, at least one command queue is created. These command queues are used to submit commands (e.g. write data, execute kernel) to the devices.

kernels are functions that are submitted and executed on OpenCL devices. Once a kernel is submitted, a global index space is defined. This index space is called NDRange. Each point in the NDRange is executed by an instance of the kernel called a work-item. Work-items are grouped into work-groups. Synchronization between

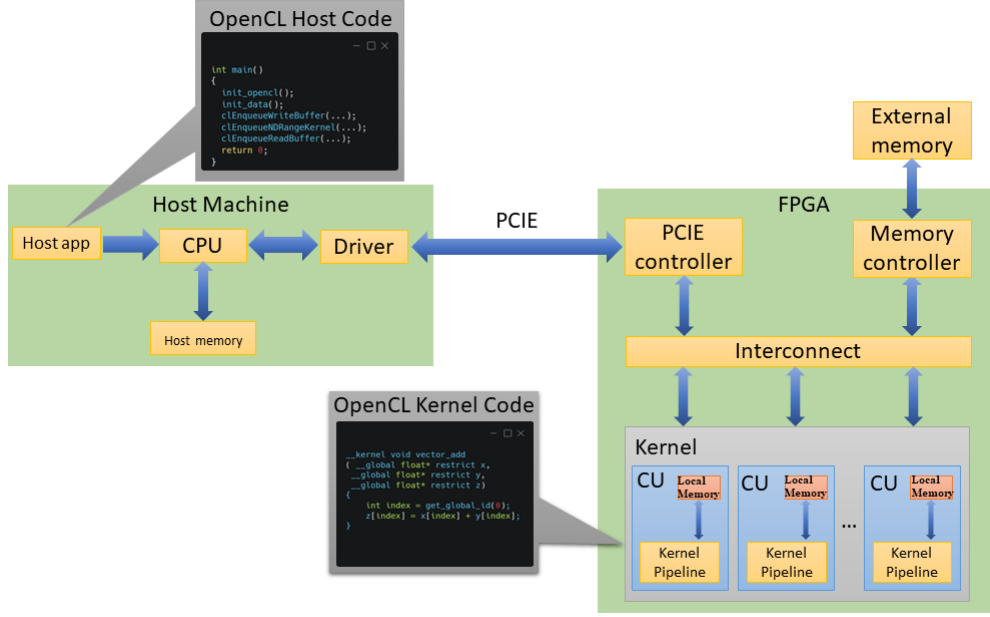


Figure 1.3: Execution model of OpenCL on FPGA.

work-items happens only within work-groups [5]. Figure 1.3 shows the execution model of OpenCL on FPGAs.

The OpenCL memory model defines five types of memories: host memory, global memory, constant memory, local memory and private memory. With the exception of host memory, all other memories resides in the OpenCL device. The Host memory is only accessible by host machine. Data on the host memory must be transferred to the global memory of the device to access it. Both Global memory and constant memory are accessible by host machine and work-items. Constant memory is a read only memory. A work-group shares data through its own local memory. Each work-item of a work-group can access its local memory. However, a work-item of a work-group can not access local memory of other work-groups. Private memory is only accessible by the work-item assigned to it [6]. Figure 1.4 shows the memory model of OpenCL.

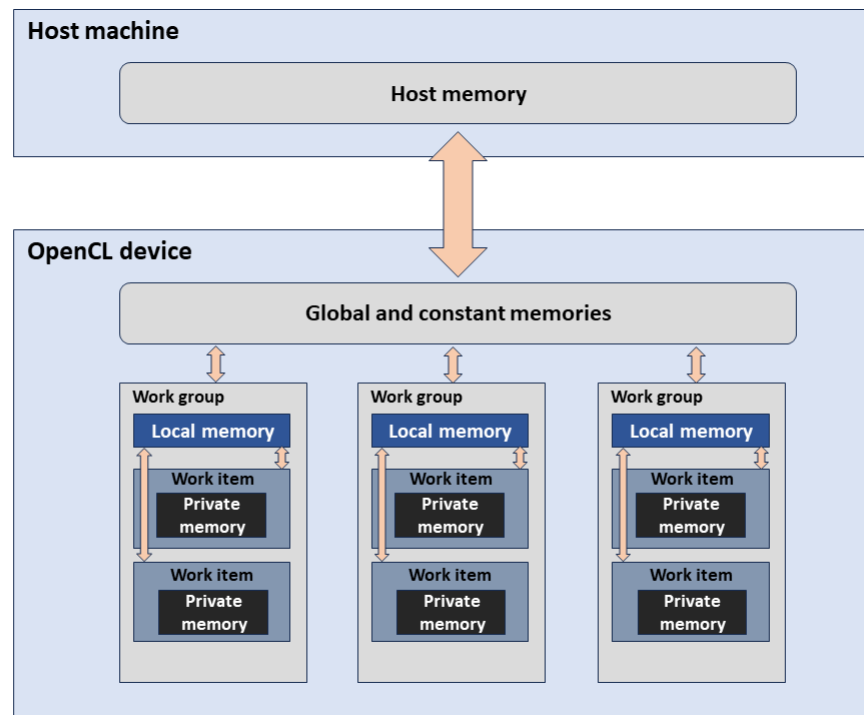


Figure 1.4: Memory model of OpenCL.

CHAPTER 2

LITERATURE REVIEW

Stencil computations are known for their data reusability. In addition, each element in a Stencil grid is independently computed, which makes them especially appealing in the context of parallel computing. Therefore, many attempts have been made in the literature to optimize the implementation of Stencil kernels on different platforms. However, there are multiple challenges that prevent exploiting these properties, causing degradation in the efficiency of any implementation of stencil. Large stencil grids, high data dependencies, and difficult data access patterns are examples of the challenges that impose a burden on the bandwidth-limited and capacity-limited memories of several platforms. In addition, most scientific applications utilize floating points which is computationally expensive.

Due to their hardware configurability, fine-grain parallelism, and power efficiency, FPGAs have been increasingly utilized as accelerators in many applications. FPGAs, however, suffer from their own limitations such as relatively lower speed and limited on-chip local memory. Many researchers investigated different implementations of Stencil kernels on FPGAs attempting to optimize the performance of Stencil compu-

tations and eliminate any existing bottlenecks.

Previously, FPGA circuitry were designed using hardware description languages (HDLs) such as Verilog and VHDL where the intended design is described at the register transfer level (RTL). This required a high level of expertise and took a long time for development. The emergence of high-level synthesis (HLS) compilers has alleviated the development time and eased the complexity of FPGA design by generating the RTL design automatically using commonly known languages such as C, C++, and OpenCL [7],[8]. However, HLS compilers have the drawback of imposing some restrictions on hardware design which might prevent the full exploitation of FPGA’s capabilities and reduce the overall performance. For example, in some HLS languages, it is not possible to specify the exact bit width for a signal. Only limited datatypes with specific bit widths are provided[9].

2.1 Data Locality and Computation Parallelism

To improve data locality and exploit parallelism, Stencil’s grids can be blocked in space, time, or both. Blocking data in space is commonly known as spatial blocking, where input grids are spatially divided into smaller blocks. It enables huge data to be processed by decomposing it into smaller blocks that fit into memory. On the other hand, time blocking (known as temporal blocking) reduces memory access by computing multiple iterations of each block before sending it back to the memory. To realize temporal blocking, Nacci et al.[10] showed that multiple iterations can be combined into single iteration. This means that if two iterations are combined, data

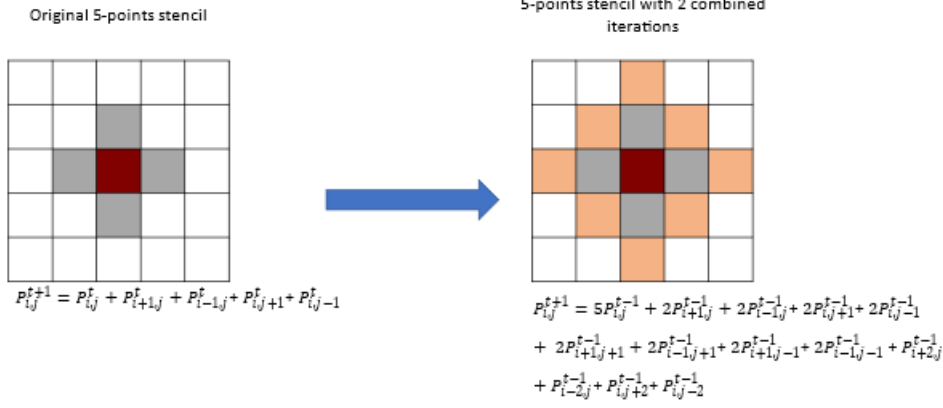


Figure 2.1: An example of combined iterations in 5-points stencil.

in iteration $t+2$ will be computed in terms of data in iteration t .

Figure 2.1 shows how the equation of 5-points Stencil is changed when 2 iterations are combined. Note that the neighbouring set is increased after combining the iterations. Another approach is to pipeline the computation of sequential iterations [7], [11]–[12]. Data computed by some processing element at specific iteration will be forwarded to another processing element to compute the next iteration. Figure 2.2 shows how each iteration is assigned to a processing element. Although this approach might be slower and requires more computation when compared to the previous one, it enhances iteration-level parallelism and improves scalability.

Realizing spatial blocking creates the problem of halo region (or ghost region). Data at the boundary of a block cannot be computed since it requires data from other neighbouring blocks. This means that after processing one iteration of a block, data at the boundaries will become invalid. The area composed of invalid data is called halo region. Combining both spatial and temporal blocking techniques enlarge the halo region. The size of halo region increases proportional to the width of temporal blocking.

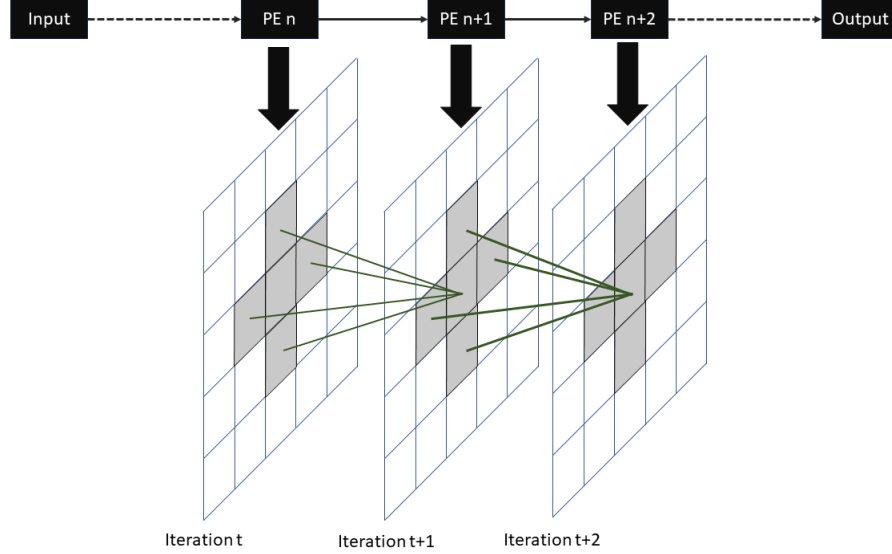


Figure 2.2: Multiple stencil iteration processing using pipeline of processing elements.

To alleviate the effect of halo region, Wang et al. [13] proposed to communicate data at the boundaries between adjacent concurrent blocks. This approach eliminates redundant data and computations but at the cost of synchronization overhead. Nate et al. [14], on the other hand, implemented a single chain of modules called Streaming Stencil Time-step (SST) connected together in a pipelined manner. Each SST computes a single time step of the array and forwards the output to the next SST. This deeply pipelined design merely exploits temporal parallelism. The spatial parallelism was exploited in the deeply pipelined design in [7], [15], [16] by vectorizing the computation in each iteration. This means that in each iteration a number of consecutive elements are computed concurrently. This design was extended to support multiple FPGAs in [11], [12] to enhance parallelism in both spatial and temporal domain. However, this creates the problem of communication bottleneck between different FPGAs.

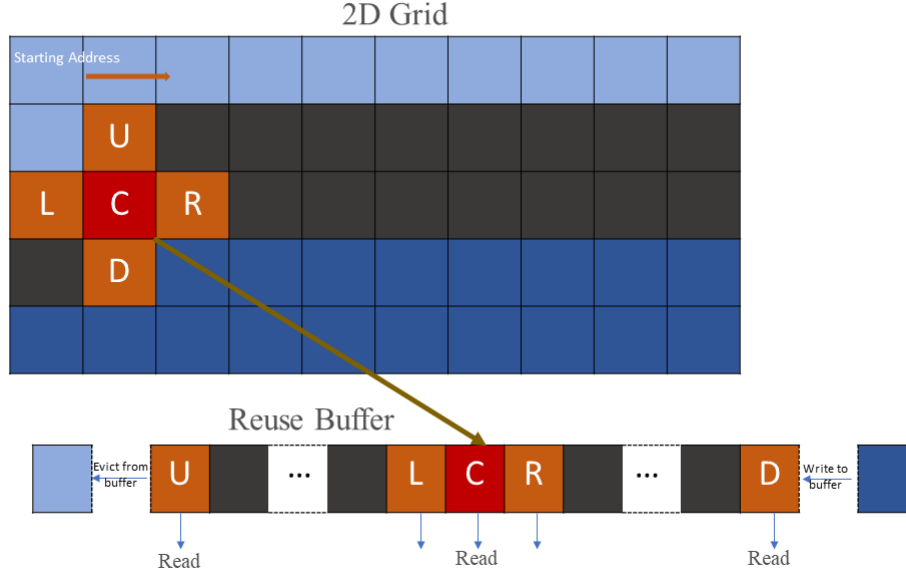


Figure 2.3: Sliding window buffer.

2.2 Data buffering

Data buffering is a key factor in the performance of Stencil computations. Hence, it was intensively investigated in the literature. Previously, partial buffering was used where only the needed data at current operation is buffered to save memory resources. However, due to the increased resources in modern FPGA, full buffering techniques currently are widely used [17]. In full buffering, data stored in the buffer is not evicted until the last computation depending on it finishes. This will exploit data reusability of stencil and reduce memory access. Sliding window buffer (or data reuse buffer) is a common buffer design realizing full buffering technique. In this design, all data between and including the first and the last needed element in the current computation is buffered. Figure 2.3 illustrates the sliding window buffer and how it is mapped to the shift register.

Many papers have attempted to implement and optimize the sliding window buffer for stencil applications. Among these papers, work in [18] proposed a non-uniform memory partitioning of the reuse buffer. They proved that their design requires the least amount of memory resources while achieving full data reuse for a single processing element. Zohouri et al. [7] have relied on the Altera OpenCL compiler to synthesize the reuse buffers. The compiler simply replicated the buffer to support the concurrent accesses. This has created redundant data and consumed more memory resources. Chi et al. [16], on the other hand, have designed their buffer using multiple FIFOs and FFs. Their buffer design feeds multiple processing elements at the same iteration. They have proved the optimality of their design. Du et al. [8] have extended the sliding window buffer to be scalable along two spatial dimensions instead of one dimension.

2.3 Floating Point Numbers

Floating point numbers are widely used in scientific applications due to their high dynamic range and good accuracy. However, their operations are complex and require many cycles to be performed. There exist some papers which implemented a floating-point unit on FPGA. Sano et al. [19] have implemented floating point with pipelined data-path in each processing element of stencil. It is executed in eight stages starting from the instruction fetch (IF). It has its own assembly language. Implementing such a unit consumes FPGA's resources. Other works, on the other hand, have attempted to find more efficient numbering format. Singh et al. [20] investigated three different number schemes and study the relation between the precision and bit width of each

scheme on stencil application. In [21], Medeiros et al. analysed the effect of converting input data of RTM algorithm from floating point numbers to low precision fixed point numbers. They studied the Signal-to-noise ratio (SNR) and the universal image quality index (UIQI) of the resultant seismic images and compared them. Most of such works are domain specific and do not generalize well. Some FPGAs such as Arria 10 includes Floating-Point Units (FPUs) on their DSP blocks [22]. Some papers have exploited this unit for their floating-point operations [12], [23]. The advantage of utilizing FPUs is to mitigate consuming high resources for floating point operations.

2.4 Stencil Implementations on FPGA

Many attempts were done in the literature to implement an efficient stencil computation in the FPGA. Among them, the work in [13] has proposed a stencil framework which realize both temporal and spatial blocking techniques. To eliminate redundant computations caused by overlapped blocks, data at the boundaries are communicated through Pipes between adjacent blocks. Pipes are OpenCL feature which is used as a communication channel between different kernels. They are inferred as FIFOs utilizing BRAMs in the FPGA. A scheduling strategy was used to reduce data communication latency. This strategy prioritizes inner data of a block in computations while at the same time waits for the updated data at the boundary. A heterogeneous tiling approach was adopted instead of the equal tiling to distribute the workload between kernels. Temporal blocking was realized in this work by combining multiple iterations. Although redundant data and computations were eliminated in this

work, synchronization overhead added to the design limits the performance. In addition, implementing the schedule strategy and the heterogeneous tiling consumes the resources of the FPGA. Another work was done in [7] with the aim of providing a parametrized stencil accelerator with unrestricted input size using OpenCL. As opposed to the previous work, this work avoided data synchronization between blocks. Instead, overlapped blocks was used for spatial blocking. A single work-item kernel was used to infer the shift registers and avoid synchronization between blocks imposed by the NDRange kernel in the OpenCL. A sliding window buffer was used to stream the data to the processing elements (PE). Temporal blocking was realized using a deeply pipelined structure. Some loop optimizations were introduced to the code to save resources and increase the operating frequency. This work was later extended in [24] to support high order stencils.

CHAPTER 3

PROPOSED FRAMEWORK

As was mentioned before, tiling (i.e. spatial blocking) of large 3-D grids is essential due to the limited capacity of FPGAs' local memories (embedded SRAMs). Two options arise; 1) Tiling the grid on the host CPU then sending them to the FPGA to perform the stencil computation then receive the computed tiles and reconstruct (untile) the grid, compute the boundary conditions, save the grid to the disk, and repeat, or 2) Send the whole initial grid to the FPGA board's RAM, perform the tiling from there, compute the next iteration(s) and boundary conditions on the FPGA, then send the whole computed grid (including boundary conditions) every few iterations back to the host to be saved in the disk. The latter option requires the FPGA to perform the tiling, computations on the main grid as well as the boundary conditions, and writing the finished results back to the host. This does not only consume more of the finite FPGA resources, but also effectively reduces its RAM's bandwidth. Though both approaches would bare communication overhead between the FPGA and the host CPU, the first approach would take advantage of the host CPU's superior memory capacity and throughput and leverage some of its computing power to compute the bound-

ary conditions. In this work, and following the first approach, a fully asynchronous pipeline for accelerating 3-D high-order stencil computations on FPGAs have been developed.

The proposed framework consists of the following fully asynchronous stages working in parallel: a tiling stage and a sending stage, both on the host CPU, an on-FPGA processing (stencil-computing) stage, a receiving stage and an untiling (grid reconstruction) stage, again on the host CPU. These stages were optimized to balance the pipeline and achieve maximum throughput. Figure 3.1 illustrates stages of the framework and Figure 3.2 shows how the 5-stage framework pipeline works.

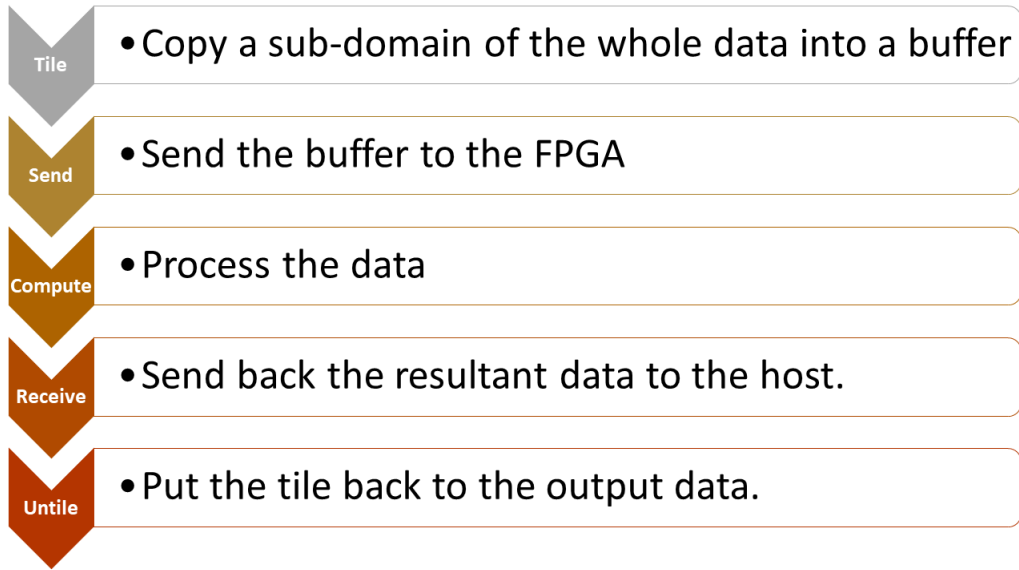


Figure 3.1: Framework stages.

The 8th order 25-point 3-D floating point stencil used in this work is shown below:

$$P_{i,j,k}^{t+1} = O_{ii}^8 (P_{i,j,k}^t) + O_{jj}^8 (P_{i,j,k}^t) + O_{kk}^8 (P_{i,j,k}^t). \quad (3.1)$$

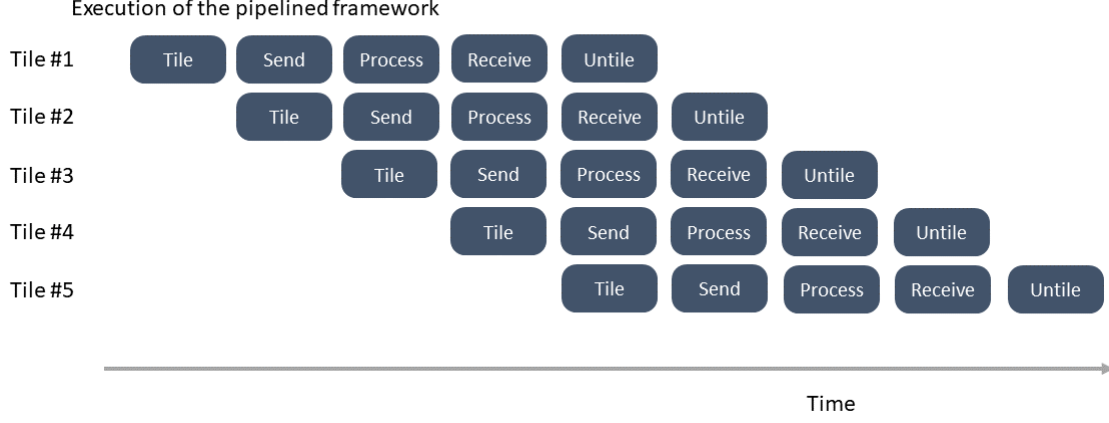


Figure 3.2: Execution of the framework pipelined stages.

where $O_{ii}^8(P_{i,j,k}^t)$ represent the 8^{th} -order approximation of the second derivative in space at the point $P_{i,j,k}^t$ along index i and it is defined as :

$$O_{ii}^8(P_{i,j,k}^n) = -\frac{205}{72}P_{i,j,k}^n + \frac{8}{5}(P_{i+1,j,k}^n + P_{i-1,j,k}^n) - \frac{1}{5}(P_{i+2,j,k}^n + P_{i-2,j,k}^n) \\ + \frac{8}{315}(P_{i+3,j,k}^n + P_{i-3,j,k}^n) - \frac{1}{560}(P_{i+4,j,k}^n + P_{i-4,j,k}^n). \quad (3.2)$$

The same goes for $O_{jj}^8(P_{i,j,k}^t)$ and $O_{kk}^8(P_{i,j,k}^t)$ along axes j and k , respectively.

This stencil is widely used to compute 2nd-order partial derivative in 3-D spaces. Figure 3.3 shows the shape of stencil used in Equation (3.1) along all indices. Compilers usually store 3-D arrays as Z-dimension (called the *fast dimension*) first, then Y-dimension (*intermediate dimension*), then X-dimension (*slow dimension*). Throughout this paper Z, Y, X axis and *fast*, *intermediate*, *slow* dimensions are used interchangeably.

The framework is composed of two main partitions: *the stencil kernel* and *host application*.

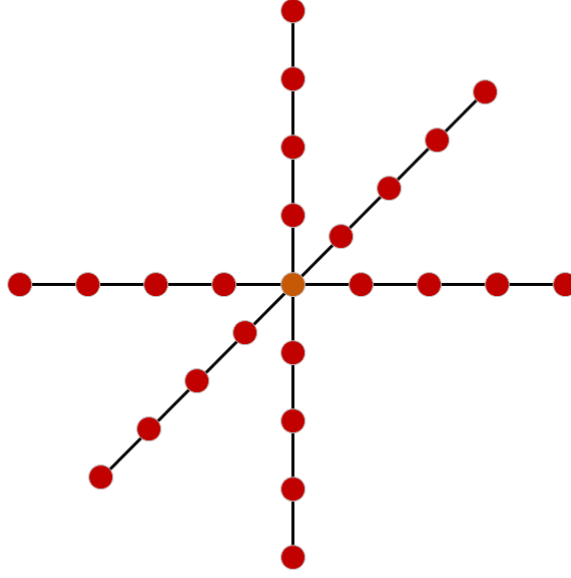


Figure 3.3: 25-point stencil shape.

3.1 Stencil Kernel

3.1.1 SODA Framework

The stencil kernel of the framework is generated using SODA (Stencil with Optimized Dataflow Architecture) automation framework [16]. SODA provides a domain-specific language (DSL) that allows the user to specify the stencil pattern and the design parameters. SODA automation framework takes the DSL from the user as an input and generates an efficient stencil implementation as an OpenCL kernel file.

The microarchitecture of SODA is composed of multiple pipelined stages. Each stage consist of a reuse buffer connected to multiple processing elements (PE). In SODA, data reuse buffers design fully exploits data reusability of stencil and minimizes local memory usage of FPGA. Figure 3.4 shows the design of a SODA reuse buffer. A

reuse buffer is composed of multiple reuse chains implemented as First In First Outs (FIFO) and Flip Flops (FF). The reuse chains have multiple tabs to forward data in different locations to the stencil PEs as specified by the specific stencil's access patterns. Hence, once a grid point has all its neighboring stencil points available in the reuse buffers, the PE can compute the next iteration's value. Data points move through the reuse buffer till they are no longer needed. Data is streamed from the FPGA board's RAM to the reuse buffer of the first stage. The reuse buffer will, then, feed the appropriate points to the PEs of the stage at the appropriate time. The output of the PEs will be forwarded to the next stage. The last stage forwards its output to the RAM. Figure 3.5 shows the microarchitecture of SODA.

SODA defines three design parameters in its DSL: input size, unroll factor, and iterate factor. It also requires all dimensions of the input grid to be pre-determined except for the slow dimension which is determined at run time as data are streamed along this dimension (as x-y planes). The unroll factor determines the number of PEs in each stage. The iterate factor indicates how many iterations are implemented in the design. Listing 3.1 shows an example of a SODA DSL file.

```

1 kernel: Jacobi
2 input float: in(128, *)
3 unroll factor: 4
4 iterate: 2
5 output float: out(0,0) = (in(0,-1) + in(-1,0) +

```

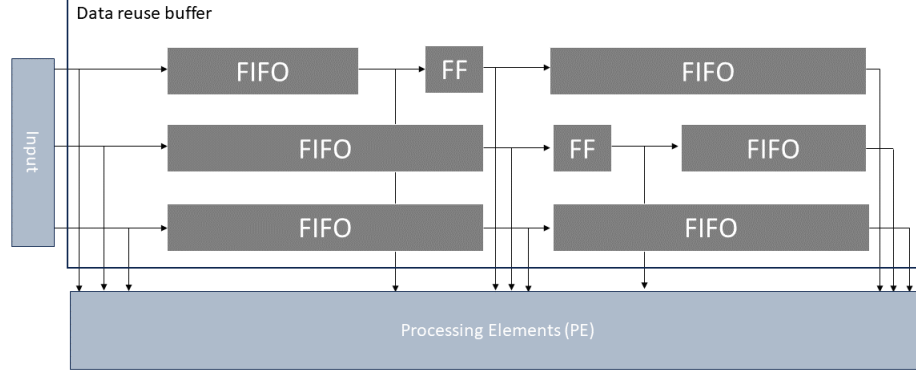


Figure 3.4: Design of SODA reuse buffer.

6

```
in(0,0) + in(1,0) + in(0,1)) * 0.1f
```

Listing 3.1: An example of SODA DSL file of 2-dimensional jacobi kernel.

3.1.2 Coalesce Memory Transfers and Kernel Execution

Kernel execution and data transfers are overlapped using a double buffering technique suggested by Intel’s documentations. In this technique, four buffers are allocated on the FPGA (i.e. two input buffers and two output buffers). Two buffers will be used by the executing kernel while the other two buffers will be used by data transfers simultaneously. In addition, three command queues are created to ensure independent submission of sending, receiving and kernel execution commands. These commands are invoked using non-blocking OpenCL functions. Synchronization between these commands are taken care by the OpenCL runtime environment using events. An event in OpenCL is an object that shows the state of a command queued to the

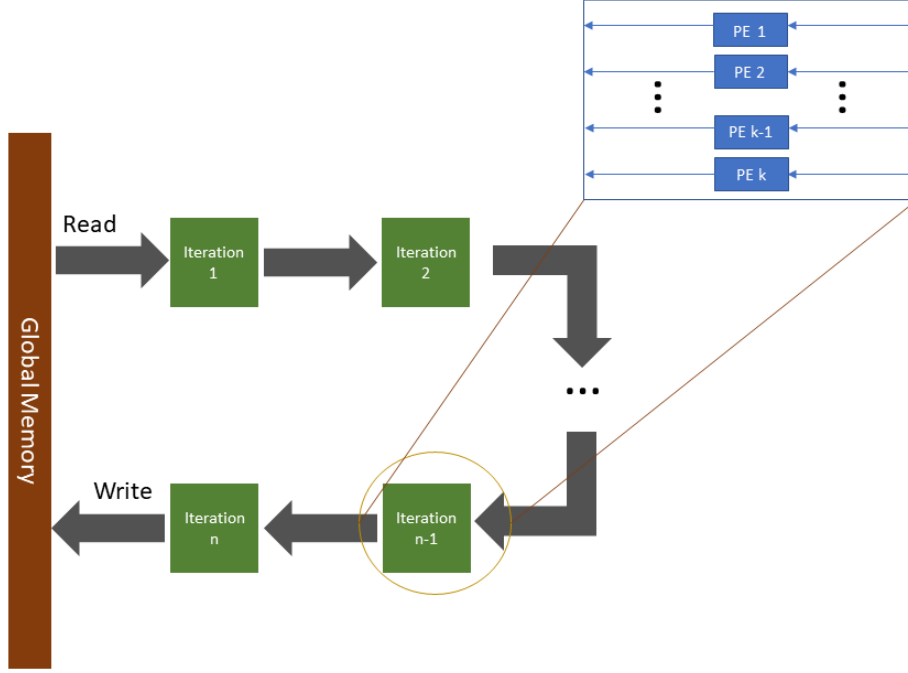


Figure 3.5: Overview of SODA microarchitecture.

device. Figure 3.6 illustrates the double buffering technique. Note that this technique requires the on-board RAM to have more than one channel.

3.2 Host Application

A special host application was developed to implement the asynchronous pipeline and orchestrate the operation of the different stages. Using OpenCL APIs to communicate with FPGAs, it spawns a thread called *device thread* for each FPGA board attached to the host. A *device thread* is responsible for tiling and untiling the stencil grid (or sub-grid in case of multiple FPGA boards) and communicating with the FPGA. Each *device thread* spawns multiple *tiling* and *untiling* threads that are assigned (i.e. pinned) to a different physical cores within the same host socket to maximize their

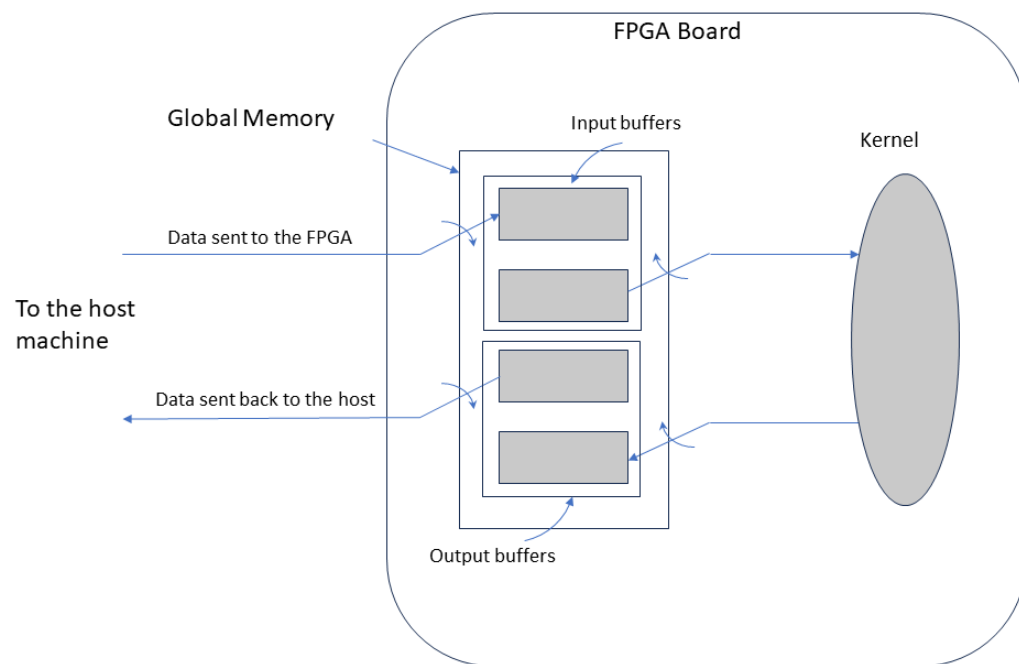


Figure 3.6: Illustration of the double buffering technique on the FPGA.

performance. Each *device thread* starts its execution by calculating the number of tiles and their addresses (in host's RAM) in sub-grid memory buffer assigned to it, and then fills a task queue with tiles' addresses. The *tiling* threads then start grabbing tiling tasks from the queue, performs tiling of the specific block (i.e. copying it to a RAM buffer), and then signals the *device thread* with the location and the index of the tile. The *device thread*, then issues non-blocking commands (API) for sending the tile to the FPGA where it is processed, and then receive the results back. Once a tile is received back, the *device thread* signals the *untiling threads* with the index and location of the tile. *Untiling threads* finally untiles the resultant tiles to the output buffer that holds the whole sub-grid. Figure 3.7 shows the execution model of single device thread.

Tiling and untiling processes are overlapped with memory transfers with the FPGA using multiple (common) buffers combined with simple asynchronous signaling to avoid hard synchronization delays (that are incurred by thread barriers or data locks). Each buffer has two 1-bit associated flags: *ready* and *valid*. *Ready* flags denote buffers that are ready to be used by *producer* threads (e.g. tiling or receiving computed tiles from the FPGA) while *valid* flags denote buffers that have valid data for *consumer* threads (e.g. tiling or sending threads). Whenever a producer needs to produce data, it scans the buffers in order looking for an unused buffer (with a *ready* flag = 1), resets its *ready* flag and proceeds to writing to that buffer. As soon as it finishes, it sets the buffer's *valid* flag. Consumers, on the other hand, scan buffers (in order) for anyone flagged as *valid*. Once a valid buffer is found, its valid flag is reset and its data is consumed. When a consumer is finished moving the buffer's data, it sets its *ready*

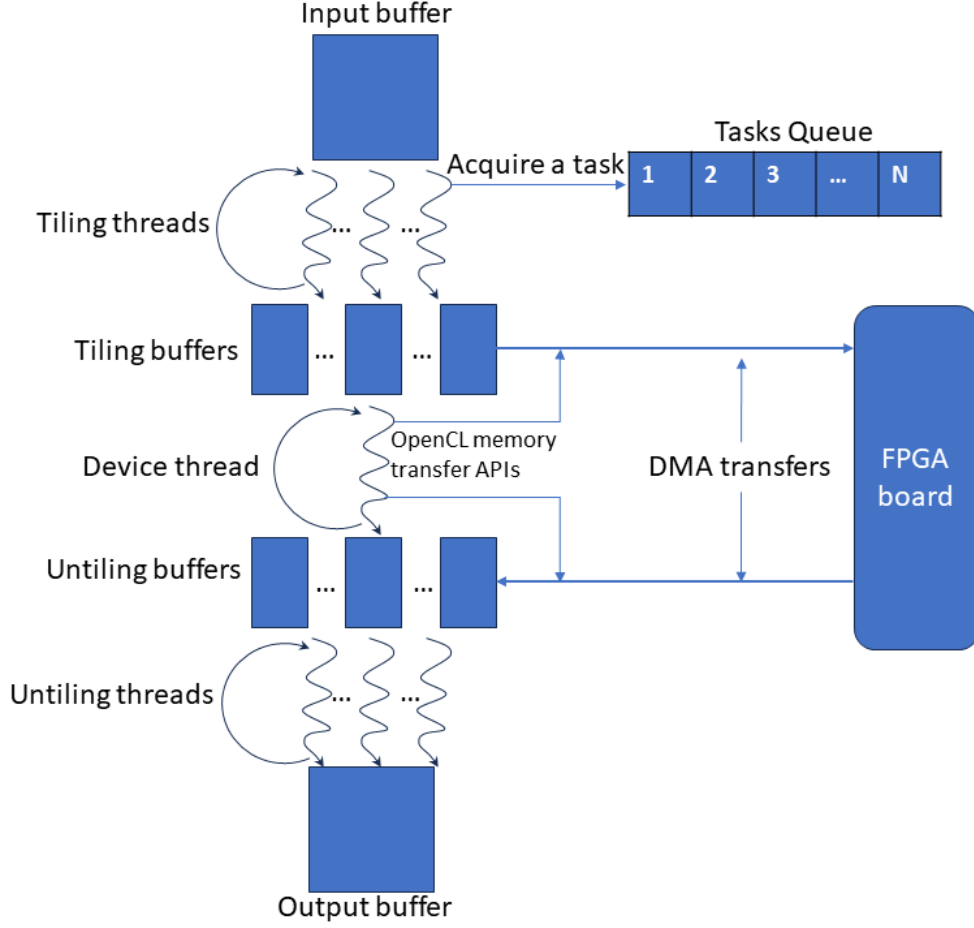


Figure 3.7: The proposed framework.

flag. Figure 3.9 illustrates these processes with 4 buffers.

3.2.1 Overlapping tiles

In order to overcome the Halo region problem, the 3-D grid is divided into overlapping tiles (only along the Z and Y directions as tiles are streamed in the X dimension). The amount of overlap points that should be added around a tile depends on the stencil length (d), the total grid (domain) size (G), and the number of combined or fused iterations (T_s). Assuming a 3-D grid of size $\mathbf{G}$ (i.e. $\mathbf{G} \times \mathbf{G} \times \mathbf{G}$) divided into equal 3-D tiles of dimensions $\mathbf{L} \times \mathbf{L} \times \mathbf{G}$, then $\mathbf{n} = \mathbf{G}/\mathbf{L}$ is the number of tiles along the Z

or Y dimensions. Figure [3.8](#) illustrates the overlap regions between adjacent tiles due to fusing several iterations. The total number of redundant points computed due to overlapping between adjacent tiles (note that the overlapped regions are computed twice and that there are no overlap regions along outer grid boundaries):

$$Redundant\ Points = 4(n - 1)G^2 \sum_{i=1}^{Ts-1} (i * d) \quad (3.3)$$

The total number of grid points (including outer boundaries) can be computed as follows:

$$Ts * G^3 + 6 * G^2 \sum_{i=1}^{Ts-1} (i * d) \quad (3.4)$$

For an 8th-order 25-point 3-D stencil ($d = 4$), and a grid size $G = 1,024$, a tile size $L = 256$, and 5 fused iterations (i.e. $Ts = 5$), the redundant points represent ~15% of the total computed points. Even though this is a significant reduction in the effective throughput, it is still better than the alternative which would require storing the computed (east and south) boundaries of each tile on the FPGA local memory or board RAM to be used by subsequent tiles. Just to illustrate the amount of buffering that would be required to store computed tiles' boundaries, let us assume that tiling starts from the north-west end of the grid and proceed row-wise eastward. The computed east boundary of a tile will become the west boundary for the next tile and the south boundaries of a row of tiles will become the north boundaries for the next row's tiles. Saving these boundaries would effectively reduce the redundant point calculations by 50% (or ~8% of the total computed points for the above example). Only one buffer will be required for the east boundary (to be overwritten by each

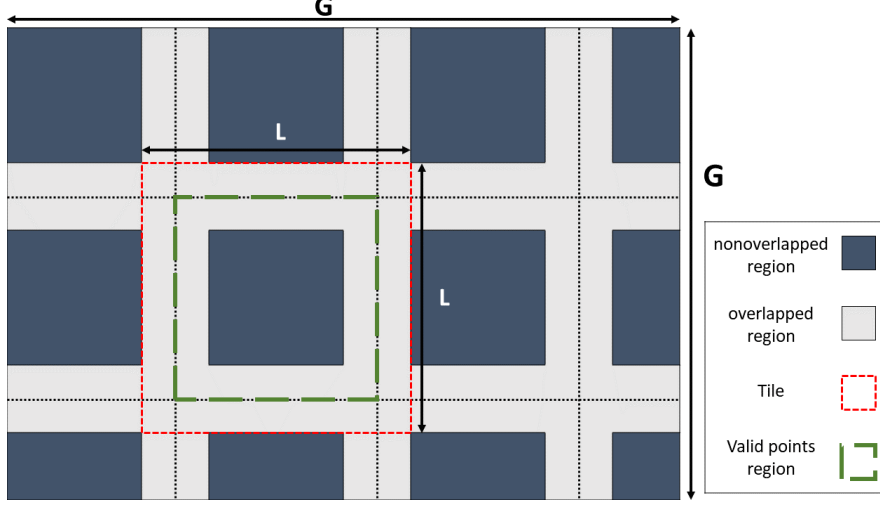


Figure 3.8: Overlapped tiles.

subsequent tile), but n buffers will be required for the south boundaries. Thus, the total required buffer size would be:

$$Buffer\ Size = 4(n + 1) * L * G * \sum_{i=1}^{Ts-1} (i * 2d) \quad (3.5)$$

Using the same parameters as in the above example, the required buffer size would be ~400 MB which cannot fit on the FPGA's local memory. Storing these boundaries on the FPGA board's RAM (global memory) would adversely impact the main stencil kernel's speed as it would compete for global memory bandwidth with the circuits that read the overlap buffers (each iteration's compute engine would be reading the relevant boundary data while the 1st iteration's engine is reading the tile's data).

3.2.2 Extending the Framework to Multiple FPGA Devices

The developed framework was extended to support two host sockets each with an attached FPGA device, all working in parallel. Multiple FPGAs attached to the

same socket won't provide extra speed unless there are direct connections between the FPGAs, something that we could not explore with the commodity server used in this work. In the case of a single FPGA, the device thread loads the whole input grid into its input buffer and starts processing it. For 2 FPGA devices, the input grid is divided into 2 overlapped sub-grids, each assigned to a different CPU socket. The host application loads one sub-grid into each socket's input buffer. At the beginning of each step, data at boundary of both sub-grids are exchanged by a dedicated thread. This thread works simultaneously with devices threads. During data exchange, tiles at the boundaries of both devices are locked with a barrier to avoid processing invalid data. Computations of other tiles proceed as usual. Since the boundary exchange time is much smaller than the processing time of the other tiles, its delay is completely masked. However, it does affect other delays via memory contention and cache pollution. Figure [3.10](#) shows the boundaries exchange process.

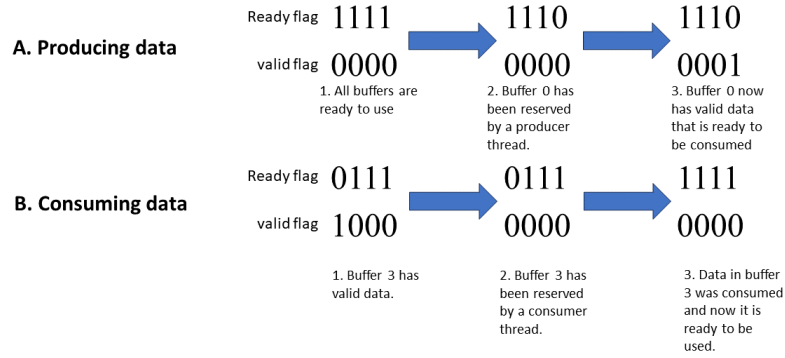


Figure 3.9: Execution example of multiple buffering technique in case of: A.producing data B. Consuming data.

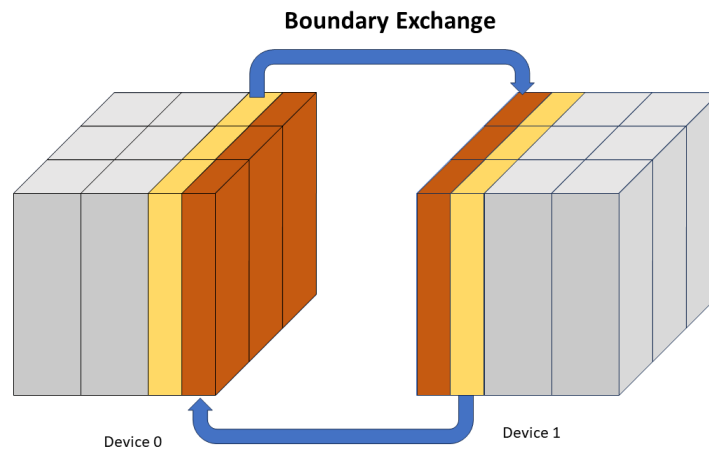


Figure 3.10: Boundary exchange between the two sub-grids: boundary tiles are shown in yellow colour, while the exchanged boundary points are shaded in brown.

CHAPTER 4

EXPERIMENTAL RESULTS

4.1 Experimental Setup

Experiments were performed on a single machine with the specifications shown in the Table 4.1. Stencil kernels were implemented on Intel’s Stratix 10 SX Development Board. The board has DDR4 memory running at 1,333 MHz. Due to the double data rate that translates to 2,666 Mega transfers per seconds. Although the board has four channels, Intel Board Support Package (BSP) only utilizes two of them. Each of the two channels has a 64-bit (8-bytes or two 32-bit words) data width. The FPGA board is attached to the host machine through 16-lane PCIe Gen3.

Table 4.1: Specification of the host machine.

Part	Model
CPU	Dual Sockets Intel E5-2620 @ 2.40 GHz (12 Physical Cores, 24 Logical Cores/Threads)
RAM	512 GB DDR4 (16 x 32GB DIMMs)
HDD	4x 2 TB in RAID 10 (RAID0+1) Configuration
OS	CentOS Linux 7

4.2 Stencil Kernel Evaluation

Initial experimentation was carried out to evaluate the performance of SODA generated HW implementation of stencil. Three parameters of the implementation were tuned to achieve the maximum throughput while maintaining a reasonable input grid size and a kernel that fits in the FPGA. These parameters are:

1. Size of the 3-D input grid. Since SODA implementation streams the grid along the x axis (slow dimension), only the size of the 2-D slice of the HW grid was tuned. Three different slice sizes were chosen: 128x128, 256x256, and 512x512. Larger slice size could not be fitted on a single FPGA and sizes were restricted to power of 2 mainly for convenience.
2. Number of fused time steps or the number of time steps computed for each block of data before writing the results back to the host. Iterations were tuned from 1 to 5 in this experiment.
3. The unrolling factor represents the number of points computed in parallel in each time step. Three unroll factors were used which are 1, 2, and 4.

Kernels with input grid slice size of 512x512 could not fit into the FPGA with more than two fused time steps due to their high utilization of block RAMs. So, a total of 36 kernels were compiled successfully.

For each kernel throughput in (GB/s) was measured as the follows:

$$Throughput = \frac{\text{size of the grid in GB}}{\text{Execution time in s}} \text{ GB/s} \quad (4.1)$$

Table 4.2: Throughput in (GB/s) versus Unrolling factors (UF) and number of fused time steps (Ts) for three tile sizes.

Slice size: 128x128					
UF\Ts	1	2	3	4	5
1	2.00	3.93	5.93	8.00	9.40
2	4.00	8.00	12.00	15.99	19.99
4	7.99	15.99	23.98	30.27	39.98
Slice size: 256x256					
UF\Ts	1	2	3	4	5
1	1.83	3.41	5.36	7.33	9.40
2	3.78	6.86	10.96	14.93	18.13
4	7.93	15.99	22.98	31.98	35.71
Slice size: 512x512					
UF\Ts	1	2	3	4	5
1	1.49	2.45	-	-	-
2	3.14	6.66	-	-	-
4	7.26	13.50	-	-	-

Resource utilization and frequency of each kernel were extracted from the report generated by the Intel OpenCL offline compiler after compilation. Power consumption of each kernel is measured using the Power Analyzer of Intel Quartus.

Table 4.2 shows the result of this experiments. As expected, throughput increases linearly with both the unrolling factor and the number of fused time steps, up to the global memory bandwidth (~ 40 GB/s).

The results in Table 4.2 indicates that a grid slice size of 128x128 would yield slightly better throughput at $T_s = 5$ (~ 40 GB/s versus 36 GB/s at 256x256 slice size). However, the percentage of Halo points (that would be computed twice when tiling is implemented) decreases linearly with the grid's slice size. So, at a 128x128 slice size and $T_s = 5$, Halo points represent 30% of the computed points thus effectively reducing the throughput by $\sim 30\%$ to ~ 28 GB/s). While for the 256x256 slice size, the Halo points represent 15% of the computed points, yielding an effective throughput

Table 4.3: On-FPGA Block RAMs (M20K Blocks) Usage.

Slice size: 128x128					
UF\Ts	1	2	3	4	5
1	550	997	1,444	1,891	2,338
2	680	1,256	1,832	2,408	2,984
4	922	1,738	2,554	3,370	4,186
Slice size: 256x256					
UF\Ts	1	2	3	4	5
1	1,318	2,533	3,748	4,963	6,178
2	1,448	2,792	4136	5,480	6,824
4	1,722	3,338	4,954	6,570	8,186
Slice size: 512x512					
UF\Ts	1	2	3	4	5
1	4,390	8,677	-	-	-
2	4,520	8,936	-	-	-
4	4,794	9,482	-	-	-

of ~ 31 GB/s. Hence a tile size of $256 \times 256 \times X$ (X is the slow dimension which is not limited to any value except by the board's RAM size) is actually much better.

Table 4.3 shows a comparison between the different kernels' block RAMs usage. It shows that BRAM utilization increases linearly with the number of fused (or chained) time steps due to the replicated re-use buffers for each time step.

Table 4.4, Table 4.5, Table 4.6, and Table 4.7 show resource utilization of ALM, DSP, MLAB, and registers on FPGA. It is observed that resource utilization increases linearly with the number of fused.

Table 4.8 shows the obtained frequencies for the three grid sizes with different unrolling factors and number of fused time steps. The target frequency was set to 500 MHz and area to minimum. The heuristics-based Intel Place and Route tool exerts best effort trying to minimize area and reach the target frequency. The obtained frequencies are mostly within 10% of the target. However, kernels with tile size of

Table 4.4: Adaptive Logic Module (ALM) usage in thousands.

Tile size: 128x128					
UF/Ts	1	2	3	4	5
1	25.60	46.00	66.38	86.72	107.09
2	40.96	76.23	111.51	146.78	182.03
4	71.96	137.72	203.36	269.04	334.70
Tile size: 256x256					
UF/Ts	1	2	3	4	5
1	27.64	50.12	72.56	95.05	117.56
2	43.17	80.46	117.84	155.28	192.61
4	74.29	142.22	210.31	278.35	346.30
Tile size: 512x512					
UF/Ts	1	2	3	4	5
1	36.28	67.31	-	-	-
2	51.92	98.11	-	-	-
4	82.58	158.92	-	-	-

Table 4.5: Digital Signal Processing (DSP) blocks Usage.

Tile size: 128x128					
UF/Ts	1	2	3	4	5
1	25	50	75	100	125
2	50	100	150	200	250
4	100	200	300	400	500
Tile size: 256x256					
UF/Ts	1	2	3	4	5
1	25	50	75	100	125
2	50	100	150	200	250
4	100	200	300	400	500
Tile size: 512x512					
UF/Ts	1	2	3	4	5
1	25	50	-	-	-
2	50	100	-	-	-
4	100	200	-	-	-

Table 4.6: Memory Logic Array Block (MLAB) usage.

Tile size: 128x128					
UF/Ts	1	2	3	4	5
1	28	46	64	82	100
2	57	93	129	165	201
4	170	306	442	578	714
Tile size: 256x256					
UF/Ts	1	2	3	4	5
1	28	46	64	82	100
2	57	93	129	165	201
4	106	178	250	322	394
Tile size: 512x512					
UF/Ts	1	2	3	4	5
1	28	46	-	-	-
2	57	93	-	-	-
4	106	178	-	-	-

Table 4.7: Registers usage in thousands.

Tile size: 128x128					
UF/Ts	1	2	3	4	5
1	71.72	125.65	180.36	235.09	289.89
2	112.98	207.33	301.64	395.85	490.24
4	197.41	376.16	553.70	731.86	910.54
Tile size: 256x256					
UF/Ts	1	2	3	4	5
1	77.75	138.15	200.32	261.93	322.07
2	119.23	221.48	320.37	420.34	522.95
4	205.40	387.97	573.40	755.14	942.57
Tile size: 512x512					
UF/Ts	1	2	3	4	5
1	113.11	209.68	-	-	-
2	153.82	285.84	-	-	-
4	230.31	453.78	-	-	-

Table 4.8: Obtained on-FPGA frequencies versus Unrolling factors (UF) and number of fused time steps (Ts) for three grid sizes.

Tile size: 128x128					
UF/Ts	1	2	3	4	5
1	500	491.67	494.44	500	470
2	500	500	500	500	500
4	500	500	500	473.33	500
Tile size: 256x256					
UF/Ts	1	2	3	4	5
1	456.67	426.67	446.67	458.33	470
2	472.22	429.17	456.67	466.67	453.33
4	495.83	500	479.17	500	446.67
Tile size: 512x512					
UF/Ts	1	2	3	4	5
1	372.22	306	-	-	-
2	392.5	416.67	-	-	-
4	454.17	422.22	-	-	-

512×512 obtained low frequencies due to the high resource utilization.

Table 4.9 shows the average power consumption for the same two grid sizes, unrolling factors and number of fused time steps. The power is dominated by the high-speed PCIe transceivers and DDR4 I/Os. As such, the grid size does not impact power significantly (doubling the grid size increases the power by $\sim 8\%$).

Table 4.10 shows the power dissipation components breakdown for a $128 \times 128 \times 1024$ grid, UF=4 and Ts=5. The actual core circuits consume 56.4% of the total power. The static device power (i.e. constant power even when nothing is running on the FPGA) is a significant 11.1% of the total power. These results also show an average power efficiency of 1.5 W/GB/s (1.5 Watts per GB/s of throughput).

In summary, the experiment shows the following:

1. On-FPGA kernel's throughput is fairly independent of the grid slice size. Hence to reduce the effect of Halo points, slice size should be increased to the maximum

Table 4.9: Average power consumption in Watts versus Unrolling factors (UF) and number of fused time steps (Ts) for three tile sizes.

Slice size: 128x128					
UF\Ts	1	2	3	4	5
1	34.33	36.34	38.94	40.91	42.33
2	35.64	38.94	42.29	45.63	48.79
4	38.13	43.81	49.43	53.50	59.56
Slice size: 256x256					
UF\Ts	1	2	3	4	5
1	35.97	39.27	43.64	48.05	52.41
2	37.21	41.14	46.81	52.00	55.97
4	39.98	47.78	54.18	61.87	64.73
Slice size: 512x512					
UF\Ts	1	2	3	4	5
1	41.63	47.91	-	-	-
2	43.23	56.07	-	-	-
4	47.07	59.64	-	-	-

Table 4.10: Power dissipation components breakdown for a grid size of 128x128x1,024, UF=4, Ts=5.

Total Thermal Power Dissipation = 59,810.93 mW	
Total Transceiver Thermal Power Dissipation = 7,476.53 mW (12.50%)	
Transceiver Standby Thermal Power Dissipation	3,164.15 mW
Transceiver Dynamic Thermal Power Dissipation	4,312.38 mW
Total I/O Thermal Power Dissipation = 11,934.75 mW (20.0%)	
I/O Standby Thermal Power Dissipation	1,787.97 mW
I/O Dynamic Thermal Power Dissipation	10,146.78 mW
Device Static Thermal Power Dissipation = 6,653.19 mW (11.1%)	
Core Dynamic Thermal Power Dissipation = 33,746.47 mW (56.4%)	

that can fit on the FPGA. The limit is the available amount of local memory (BRAM),

2. Throughput increases linearly with the unrolling factor and the number of fused (or chained) time steps up to the total bandwidth of the global memory (~ 40 GB/s in our case). An unrolling factor beyond the global memory's interface width is useless. Hence, for our FPGA which has two 64-bit wide channels, an unrolling factor of 4 (i.e. four 32-bit floats are processed in parallel) is the best. Similarly, a T_s of 5 is the best,
3. FPGA's RAM utilization is fairly independent of the unrolling factor but increases linearly with T_s . As such, the maximum grid slice size that can be fit with a T_s of 5 was 256×256 . Other resources increases linearly with both unrolling factor (UF) and fused iterations (T_s).
4. Power consumption increases slightly with the unrolling factor, but more with T_s . More than 40% of the consumed power is in interfaces (PICe and DDR4 RAM), hence doubling the grid slice size from 128×128 to 256×256 (at UF of 4 and T_s of 5) only increased the total power by less than 9%.

Hence, the optimum computing conditions for the Stratix 10 FPGA from Intel are: grid slice size of 256×256 , UF of 4, and T_s of 5. These would yield a throughput of 36 GB/s (or 31 GB/s effective throughput taking Halo effect into account) and 1.81 W/GB/s power efficiency.

4.3 SODA Modification to Reduce FPGA Resource Utilization

```
kernel
__attribute__((reqd_work_group_size(1, 1, 1)))
__attribute__((max_global_work_dim(0)))
__attribute__((uses_global_work_offset(0)))
void stencil_00_module_0_instance_0(global float* restrict bank_0_t1, ulong coalesced_data_num)
{
    for (ulong i = 0; i < coalesced_data_num; ++i) {
        float dram_t1_bank_0_buf[1];
        dram_t1_bank_0_buf[0] = bank_0_t1[i];
        write_channel_intel(from_t1_bank_0_to_t1_offset_0, dram_t1_bank_0_buf[0]);
    }
}

kernel
__attribute__((reqd_work_group_size(1, 1, 1)))
__attribute__((max_global_work_dim(0)))
__attribute__((uses_global_work_offset(0)))
void stencil_27_module_9_instance_0(global float* restrict bank_0_t0, ulong coalesced_data_num)
{
    for (ulong i = 0; i < coalesced_data_num; ++i) {
        float fifo_ref_0 = read_channel_intel(from_t0_pe_0_to_t0_bank_0);
        float dram_t0_bank_0_buf[1];
        dram_t0_bank_0_buf[0] = fifo_ref_0;
        bank_0_t0[i] = dram_t0_bank_0_buf[0];
    }
}
```

Figure 4.1: Code snippet showing the two non-autorun kernels for reading and writing from/to the global memory.

The previous results showed that, in general, throughput can be maintained at higher grid size and it scales linearly with the UF and the Ts. The FPGA limited resources, however, represent the ultimate barrier to increasing the grid size. It is observed that SODA produces many Autorun kernels for the FIFOs that make up the re-use buffer and the processing elements. An Autorun kernel is a type of Intel defined OpenCL kernel which executes automatically without being launched explicitly. To infer such a kernel by the Intel FPGA SDK for OpenCL Offline Compiler, an “autorun” attribute should be added to the kernel along with either *max_global_work_dim(0)* attribute or the *reqd_work_group_size(X,Y,Z)*. In addition, an autorun kernel must

```

kernel
__attribute__((reqd_work_group_size(1, 1, 1)))
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
void stencil_25_module_7_instance_0()
{
    ushort ptr_delay_65536 = 0;
    float fifo_ref_0_delayed_65536_buf[65536];
    #pragma ivdep
    for (; ; ) {
        const float fifo_ref_0_delayed_65536 = fifo_ref_0_delayed_65536_buf[ptr_delay_65536];
        float fifo_ref_0 = read_channel_intel(from_t1_offset_458752_to_t1_offset_524288);
        const float let_0 = fifo_ref_0_delayed_65536;
        write_channel_intel(from_t1_offset_524288_to_t0_pe_0, let_0);
        fifo_ref_0_delayed_65536_buf[ptr_delay_65536] = fifo_ref_0;
        ptr_delay_65536 < 65535 ? (++ptr_delay_65536) : (ptr_delay_65536 = 0);
    }
}

```

Figure 4.2: Code snippet of an autorun kernel of a forwarding module (FIFO).

not have any argument. The logic for communication between the host and these types of kernels are omitted which reduces the area of utilization. The only way to communicate with autorun kernels is through channels or pipes. The main advantage of autorun kernels is that their execution is decoupled from the execution of the main kernels. In SODA documentation, an autorun kernel is called a module. SODA automation framework provides four module clustering levels in its DSL (Domain Specific Language): none, fine, coarse, and full. In all clustering levels, there are two main kernels (non-autorun kernels), one for reading data from global memory and the other for writing to the global memory. Figure 4.1 shows a code snippet for both kernels. In the (none) clustering level, SODA automation framework generates an autorun kernel for each forwarding module (FIFO) and compute module (PE) in the design. Figure 4.2 and Figure 4.3 show code snippets for an autorun kernel for a forwarding module and an autorun kernel for a compute module, respectively. In the fine clustering level, all forwarding modules of each stage are fused into one autorun kernel while the compute modules are kept separate. In the coarse clustering

```

kernel
__attribute__((reqd_work_group_size(1, 1, 1)))
__attribute__((max_global_work_dim(0)))
__attribute__((autotune))
void stencil_26_module_8_instance_0()
{
    for (;;) {
        float fifo_ref_0 = read_channel_intel(from_t1_offset_261888_to_t0_pe_0);
        float fifo_ref_1 = read_channel_intel(from_t1_offset_261632_to_t0_pe_0);
        float fifo_ref_2 = read_channel_intel(from_t1_offset_261376_to_t0_pe_0);
        float fifo_ref_3 = read_channel_intel(from_t1_offset_261120_to_t0_pe_0);
        float fifo_ref_4 = read_channel_intel(from_t1_offset_262400_to_t0_pe_0);
        float fifo_ref_5 = read_channel_intel(from_t1_offset_262656_to_t0_pe_0);
        float fifo_ref_6 = read_channel_intel(from_t1_offset_262912_to_t0_pe_0);
        float fifo_ref_7 = read_channel_intel(from_t1_offset_263168_to_t0_pe_0);
        float fifo_ref_8 = read_channel_intel(from_t1_offset_262143_to_t0_pe_0);
        float fifo_ref_9 = read_channel_intel(from_t1_offset_262142_to_t0_pe_0);
        float fifo_ref_10 = read_channel_intel(from_t1_offset_262141_to_t0_pe_0);
        float fifo_ref_11 = read_channel_intel(from_t1_offset_262140_to_t0_pe_0);
        float fifo_ref_12 = read_channel_intel(from_t1_offset_262145_to_t0_pe_0);
        float fifo_ref_13 = read_channel_intel(from_t1_offset_262146_to_t0_pe_0);
        float fifo_ref_14 = read_channel_intel(from_t1_offset_262147_to_t0_pe_0);
        float fifo_ref_15 = read_channel_intel(from_t1_offset_262148_to_t0_pe_0);
        float fifo_ref_16 = read_channel_intel(from_t1_offset_196608_to_t0_pe_0);
        float fifo_ref_17 = read_channel_intel(from_t1_offset_131072_to_t0_pe_0);
        float fifo_ref_18 = read_channel_intel(from_t1_offset_65536_to_t0_pe_0);
        float fifo_ref_19 = read_channel_intel(from_t1_offset_0_to_t0_pe_0);
        float fifo_ref_20 = read_channel_intel(from_t1_offset_327680_to_t0_pe_0);
        float fifo_ref_21 = read_channel_intel(from_t1_offset_393216_to_t0_pe_0);
        float fifo_ref_22 = read_channel_intel(from_t1_offset_458752_to_t0_pe_0);
        float fifo_ref_23 = read_channel_intel(from_t1_offset_524288_to_t0_pe_0);
        float fifo_ref_24 = read_channel_intel(from_t1_offset_262144_to_t0_pe_0);
        write_channel_intel(from_t0_pe_0_to_t0_bank_0, (float)((1.6f * fifo_ref_0) + (-0.2f *
        fifo_ref_1) + (0.0254f * fifo_ref_2) + (-0.0018f * fifo_ref_3) + (1.6f * fifo_ref_4) + (-0.2f *
        fifo_ref_5) + (0.0254f * fifo_ref_6) + (-0.0018f * fifo_ref_7) + (1.6f * fifo_ref_8) + (-0.2f *
        fifo_ref_9) + (0.0254f * fifo_ref_10) + (-0.0018f * fifo_ref_11) + (1.6f * fifo_ref_12) +
        (-0.2f * fifo_ref_13) + (0.0254f * fifo_ref_14) + (-0.0018f * fifo_ref_15) + (1.6f *
        fifo_ref_16) + (-0.2f * fifo_ref_17) + (0.0254f * fifo_ref_18) + (-0.0018f * fifo_ref_19) +
        (1.6f * fifo_ref_20) + (-0.2f * fifo_ref_21) + (0.0254f * fifo_ref_22) + (-0.0018f *
        fifo_ref_23) + (-8.54167f * fifo_ref_24)));
    }
}

```

Figure 4.3: Code snippet of an autorun kernel for a compute module (PE).

level, all forwarding modules and compute modules of a stage are fused together into one autorun kernel. Figure 4.4 shows a code snippet of an autorun kernel of fused forwarding and compute modules for the Coarse clustering option. In the full clustering level, modules of all stages are fused into one autorun kernel, as shown in Figure 4.5. Still, separate kernels are used for reading/writing data from/to FPGA's board memory. This results in channel between these kernels.

The first modification was introduced to the (none) clustering level where the forwarding module code was replaced by a shift register code as suggested in Intel's documentation. Figure 4.6 shows the code snippet of the shift register code.

The second and third modifications were introduced to the (fine) and (coarse) clustering levels respectively where all unnecessary variables and structures in the code


```

__kernel
__attribute__((reqd_work_group_size(1, 1, 1)))
__attribute__((max_global_work_dim(0)))
__attribute__((autotun))
void stencil_1_module_1_instance_0()
{
    //Declaring FIFOs' pointers
    uint14_t ptr_delay_16384 = 0;
    uint14_t ptr_delay_15872 = 0;
    uint7_t ptr_delay_128 = 0;
    uint7_t ptr_delay_124 = 0;
    uint1_t ptr_delay_1 = 0;

    //Declaring the FIFOs
    float let_0_delayed_16384_buf[16384];
    float let_3_delayed_16384_buf[16384];
    .....
    .....
    .....
    float let_362_delayed_16384_buf[16384];
    float let_365_delayed_16384_buf[16384];

#pragma ivdep
    //The main loop
    for (; ; ) {
        //Caching the current values of FIFOs
        const float let_0_delayed_16384 = let_0_delayed_16384_buf[ptr_delay_16384];
        const float let_3_delayed_16384 = let_3_delayed_16384_buf[ptr_delay_16384];
        .....
        .....
        .....
        const float let_362_delayed_16384 = let_362_delayed_16384_buf[ptr_delay_16384];
        const float let_365_delayed_16384 = let_365_delayed_16384_buf[ptr_delay_16384];

        //Reading from from the input kernel
        float fifo_ref_0 = read_channel_intel(from_t1_bank_0_to_module_0);

        //Calculating the output of the first stage
        const float let_0 = fifo_ref_0;
        const float let_1 = fifo_ref_0;
        .....
        .....
        .....
        const float let_71 = let_69_delayed_16384;
        const float let_72 = let_71;
        const float let_73 = (float)((1.6f * let_22) + (-0.2f * let_19) +
        (0.0254f * let_16) + (-0.0018f * let_13) + (1.6f * let_52) +
        (-0.2f * let_55) + (0.0254f * let_58) + (-0.0018f * let_61) +
        (1.6f * let_34) + (-0.2f * let_31) + (0.0254f * let_28) +
        (-0.0018f * let_25) + (1.6f * let_40) + (-0.2f * let_43) +
        (0.0254f * let_46) + (-0.0018f * let_49) + (1.6f * let_10) +
        (-0.2f * let_7) + (0.0254f * let_4) + (-0.0018f * let_1) +
        (1.6f * let_64) + (-0.2f * let_67) + (0.0254f * let_70) +
        (-0.0018f * let_72) + (-8.54167f * let_37));

        //Calculating the output of the second stage
        const float let_74 = let_73;
        const float let_75 = let_73;
        .....
        .....
        .....
        const float let_145 = let_143_delayed_16384;
        const float let_146 = let_145;
        const float let_147 = (float)((1.6f * let_96) + (-0.2f * let_93) +
        (0.0254f * let_98) + (-0.0018f * let_87) + (1.6f * let_126) +
        (-0.2f * let_129) + (0.0254f * let_132) + (-0.0018f * let_135) +
        (1.6f * let_108) + (-0.2f * let_105) + (0.0254f * let_102) + (-0.0018f * let_99) +
        (1.6f * let_114) + (-0.2f * let_117) + (0.0254f * let_120) + (-0.0018f * let_123) +
        (1.6f * let_84) + (-0.2f * let_81) + (0.0254f * let_78) + (-0.0018f * let_75) +
        (1.6f * let_138) + (-0.2f * let_141) + (0.0254f * let_144) + (-0.0018f * let_146) +
        (-8.54167f * let_111));
        .....
        .....
        .....
        //calculating the rest of stages
        .....
        .....
        .....
        //calculating the last stage and writing it to the memory
        const float let_296 = let_295;
        const float let_297 = let_295;
        .....
        .....
        .....
        const float let_367 = let_365_delayed_16384;
        const float let_368 = let_367;
        write_channel_intel(from_module_0_to_t0_bank_0, (float)((1.6f * let_318) +
        (-0.2f * let_315) + (0.0254f * let_312) + (-0.0018f * let_309) + (1.6f * let_348) +
        (-0.2f * let_351) + (0.0254f * let_354) + (-0.0018f * let_357) + (1.6f * let_338) +
        (-0.2f * let_327) + (0.0254f * let_324) + (-0.0018f * let_321) + (1.6f * let_336) +
        (-0.2f * let_339) + (0.0254f * let_342) + (-0.0018f * let_345) + (1.6f * let_306) +
        (-0.2f * let_303) + (0.0254f * let_300) + (-0.0018f * let_297) + (1.6f * let_368) +
        (-0.2f * let_363) + (0.0254f * let_366) + (-0.0018f * let_368) + (-8.54167f *
        let_333)));
        //Writing the data to next FIFOs
        let_0_delayed_16384_buf[ptr_delay_16384] = let_0;
        let_3_delayed_16384_buf[ptr_delay_16384] = let_3;
        .....
        .....
        .....
        let_362_delayed_16384_buf[ptr_delay_16384] = let_362;
        let_365_delayed_16384_buf[ptr_delay_16384] = let_365;

        //Advancing the pointers
        ptr_delay_16384 < 16383 ? (++ptr_delay_16384) : (ptr_delay_16384 = 0);
        ptr_delay_15872 < 15871 ? (++ptr_delay_15872) : (ptr_delay_15872 = 0);
        ptr_delay_128 < 127 ? (++ptr_delay_128) : (ptr_delay_128 = 0);
        ptr_delay_124 < 123 ? (++ptr_delay_124) : (ptr_delay_124 = 0);
        ptr_delay_1 < 0 ? (++ptr_delay_1) : (ptr_delay_1 = 0);
    }
}

```

Figure 4.5: Code snippet of combined stages in SODA using the (full) clustering level.

```

kernel
__attribute__((reqd_work_group_size(1, 1, 1)))
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
void offset_16384_reg_iteration_0()
{
    float shift_reg[16384];
    #pragma ivdep
    for(;; ) {
        write_channel_intel(from_offset_16384_to_offset_32768_iteration_0,shift_reg[0]);
        write_channel_intel(from_offset_16384_to_pe_0_iteration_0,shift_reg[0]);
        #pragma unroll
        for(int i=0; i<16383; i++){
            shift_reg[i] = shift_reg[i+1];
        }
        shift_reg[16383] = read_channel_intel(from_offset_0_to_offset_16384_iteration_0);
    }
}

```

Figure 4.6: The code for the shift register replacing the forwarding module in the none clustering option.

were removed. Figure 4.7 and Figure 4.8 show code snippets of the modified (fine) and (coarse) clustering levels. These modifications were dubbed combined FIFOs and combined FIFOs and PE, respectively.

In the final modification, all kernels (autorun and non-autorun ones) were merged into a single kernel. Figure 4.9 shows a code snippet of a single kernel code. This modification will be called single kernel version throughout this experiment.

For this experiment, only two slice sizes were used: 128×128 and 256×256 . A 512×512 slice size is still impossible to place on the FPGA for a Ts more than 2. Several combinations of UF and Ts values were tried. Table 4.11 and Table 4.14 summarize the results for all clustering options (including no clustering) for the basic case of UF=1 and Ts=1 (no unrolling nor time steps fusion) for the 128×128 and the 256×256 slice sizes, respectively. These results show that using Intel's documented shift register reduces the utilization of some resources negligibly while

```

kernel AUTORUN void registers(){
    //Declaring the FIFOs
    float slow_n_4_to_slow_n_3[16384];
    float slow_n_3_to_slow_n_2[16384];
    .....
    .....
    .....
    float slow_p_2_to_slow_p_3[16384];
    float slow_p_3_to_slow_p_4[16384];

    //Declaring FIFOs' pointers
    uint16_t slow_n_4_to_slow_n_3_ptr = 0;
    uint16_t slow_n_3_to_slow_n_2_ptr = 0;
    .....
    .....
    .....
    uint16_t slow_p_2_to_slow_p_3_ptr = 0;
    uint16_t slow_p_3_to_slow_p_4_ptr = 0;
    //The main Loop
    for(;;){
        //Caching the current values of FIFOs
        float input = read_channel_intel(from_input_to_registers);
        float slow_n_4_to_slow_n_3_tmp = slow_n_4_to_slow_n_3[slow_n_4_to_slow_n_3_ptr];
        float slow_n_3_to_slow_n_2_tmp = slow_n_3_to_slow_n_2[slow_n_3_to_slow_n_2_ptr];
        .....
        .....
        .....
        float slow_p_2_to_slow_p_3_tmp = slow_p_2_to_slow_p_3[slow_p_2_to_slow_p_3_ptr];
        float slow_p_3_to_slow_p_4_tmp = slow_p_3_to_slow_p_4[slow_p_3_to_slow_p_4_ptr];

        //Sending the current values of FIFOs to the compute module
        write_channel_intel(from_offset_0_to_pe_0_iteration_0,input);
        write_channel_intel(from_offset_65536_to_pe_0_iteration_0,slow_n_4_to_slow_n_3_tmp);
        .....
        .....
        .....
        write_channel_intel(from_offset_458752_to_pe_0_iteration_0,slow_p_2_to_slow_p_3_tmp);
        write_channel_intel(from_offset_524288_to_pe_0_iteration_0,slow_p_3_to_slow_p_4_tmp);

        //Writing the data to next FIFOs
        slow_n_4_to_slow_n_3[slow_n_4_to_slow_n_3_ptr] = input;
        slow_n_3_to_slow_n_2[slow_n_3_to_slow_n_2_ptr] = slow_n_4_to_slow_n_3_tmp;
        .....
        .....
        .....
        slow_p_2_to_slow_p_3[slow_p_2_to_slow_p_3_ptr] = slow_p_1_to_slow_p_2_tmp;
        slow_p_3_to_slow_p_4[slow_p_3_to_slow_p_4_ptr] = slow_p_2_to_slow_p_3_tmp;

        //Advancing the pointers
        slow_n_4_to_slow_n_3_ptr < 16383 ? (++slow_n_4_to_slow_n_3_ptr) : (slow_n_4_to_slow_n_3_ptr =
0);
        slow_n_3_to_slow_n_2_ptr < 16383 ? (++slow_n_3_to_slow_n_2_ptr) : (slow_n_3_to_slow_n_2_ptr =
0);
        .....
        .....
        .....
        slow_p_2_to_slow_p_3_ptr < 16383 ? (++slow_p_2_to_slow_p_3_ptr) : (slow_p_2_to_slow_p_3_ptr =
0);
        slow_p_3_to_slow_p_4_ptr < 16383 ? (++slow_p_3_to_slow_p_4_ptr) : (slow_p_3_to_slow_p_4_ptr =
0);
    }
}

```

Figure 4.7: Code snippet of a kernel with combined FIFO modules.

```

kernel AUTORUN void registers(){
    //Declaring the FIFOs
    float slow_n_4_to_slow_n_3[16384];
    float slow_n_3_to_slow_n_2[16384];
    .....
    .....
    .....
    float slow_p_2_to_slow_p_3[16384];
    float slow_p_3_to_slow_p_4[16384];

    //Declaring FIFOs' pointers
    uint16_t slow_n_4_to_slow_n_3_ptr = 0;
    uint16_t slow_n_3_to_slow_n_2_ptr = 0;
    .....
    .....
    .....
    uint16_t slow_p_2_to_slow_p_3_ptr = 0;
    uint16_t slow_p_3_to_slow_p_4_ptr = 0;
    //The main Loop
    for(;;){
        //Caching the current values of FIFOs
        float input = read_channel_intel(from_input_to_registers);
        float slow_n_4_to_slow_n_3_tmp = slow_n_4_to_slow_n_3[slow_n_4_to_slow_n_3_ptr];
        .....
        .....
        .....
        float slow_p_2_to_slow_p_3_tmp = slow_p_2_to_slow_p_3[slow_p_2_to_slow_p_3_ptr];
        float slow_p_3_to_slow_p_4_tmp = slow_p_3_to_slow_p_4[slow_p_3_to_slow_p_4_ptr];

        //Calculating the output
        float output =
            c4*input
            + c3*slow_n_4_to_slow_n_3_tmp + c2*slow_n_3_to_slow_n_2_tmp +
            c1*slow_n_2_to_slow_n_1_tmp +
            c4*slow_n_1_to_med_n_4_tmp + c3*med_n_4_to_med_n_3_tmp + c2*med_n_3_to_med_n_2_tmp +
            c1*med_n_2_to_med_n_1_tmp +
            c4*med_n_1_to_fast_n_4_tmp + c3*fast_n_4_to_fast_n_3_tmp + c2*fast_n_3_to_fast_n_2_tmp +
            c1*fast_n_2_to_fast_n_1_tmp +
            c0*fast_n_1_to_center_tmp*3+
            c1*center_to_fast_p_1_tmp + c2*fast_p_1_to_fast_p_2_tmp + c3*fast_p_2_to_fast_p_3_tmp +
            c4*fast_p_3_to_fast_p_4_tmp +
            c1*fast_p_4_to_med_p_1_tmp + c2*med_p_1_to_med_p_2_tmp + c3*med_p_2_to_med_p_3_tmp +
            c4*med_p_3_to_med_p_4_tmp +
            c1*med_p_4_to_slow_p_1_tmp + c2*slow_p_1_to_slow_p_2_tmp + c3*slow_p_2_to_slow_p_3_tmp +
            c4*slow_p_3_to_slow_p_4_tmp ;

        //Sending the output to the next stage or to the memory
        write_channel_intel(from_pe_0_to_output_0_iteration_1,output);

        //Writing the data to next FIFOs
        slow_n_4_to_slow_n_3[slow_n_4_to_slow_n_3_ptr] = input;
        slow_n_3_to_slow_n_2[slow_n_3_to_slow_n_2_ptr] = slow_n_4_to_slow_n_3_tmp;
        .....
        .....
        .....
        slow_p_2_to_slow_p_3[slow_p_2_to_slow_p_3_ptr] = slow_p_1_to_slow_p_2_tmp;
        slow_p_3_to_slow_p_4[slow_p_3_to_slow_p_4_ptr] = slow_p_2_to_slow_p_3_tmp;

        //Advancing the pointers
        slow_n_4_to_slow_n_3_ptr < 16383 ? (++slow_n_4_to_slow_n_3_ptr) : (slow_n_4_to_slow_n_3_ptr = 0);
        slow_n_3_to_slow_n_2_ptr < 16383 ? (++slow_n_3_to_slow_n_2_ptr) : (slow_n_3_to_slow_n_2_ptr = 0);
        .....
        .....
        .....
        slow_p_2_to_slow_p_3_ptr < 16383 ? (++slow_p_2_to_slow_p_3_ptr) : (slow_p_2_to_slow_p_3_ptr = 0);
        slow_p_3_to_slow_p_4_ptr < 16383 ? (++slow_p_3_to_slow_p_4_ptr) : (slow_p_3_to_slow_p_4_ptr = 0);
    }
}

```

Figure 4.8: Code snippet of a kernel with combined FIFO and compute modules.

```

kernel IOKERNEL void in_kernel( global float* restrict input,
                                global float* restrict output, ulong coalesced_data_num){

    //Declaring the FIFOs
    float slow_n_4_to_slow_n_3[SLICE_SIZE];
    float slow_n_3_to_slow_n_2[SLICE_SIZE];
    .....
    .....
    .....
    float slow_p_2_to_slow_p_3[SLICE_SIZE];
    float slow_p_3_to_slow_p_4[SLICE_SIZE];

    //Declaring FIFOs' pointers
    SLICE_PTR_TYPE slow_n_4_to_slow_n_3_ptr = 0;
    SLICE_PTR_TYPE slow_n_3_to_slow_n_2_ptr = 0;
    .....
    .....
    .....
    SLICE_PTR_TYPE slow_p_2_to_slow_p_3_ptr = 0;
    SLICE_PTR_TYPE slow_p_3_to_slow_p_4_ptr = 0;
    //The main Loop
    for(ulong i = 0; i < coalesced_data_num; ++i){
        //Reading from the memory
        float in = input[i];

        //Caching the current values of FIFOs
        float slow_n_4_to_slow_n_3_tmp = slow_n_4_to_slow_n_3[slow_n_4_to_slow_n_3_ptr];
        float slow_n_3_to_slow_n_2_tmp = slow_n_3_to_slow_n_2[slow_n_3_to_slow_n_2_ptr];
        .....
        .....
        .....
        float slow_p_2_to_slow_p_3_tmp = slow_p_2_to_slow_p_3[slow_p_2_to_slow_p_3_ptr];
        float slow_p_3_to_slow_p_4_tmp = slow_p_3_to_slow_p_4[slow_p_3_to_slow_p_4_ptr];

        //Calculating the output
        output[i] =
            c4*in
            + c3*slow_n_4_to_slow_n_3_tmp + c2*slow_n_3_to_slow_n_2_tmp +
            c1*slow_n_2_to_slow_n_1_tmp +
            c4*slow_n_1_to_med_n_4_tmp + c3*med_n_4_to_med_n_3_tmp + c2*med_n_3_to_med_n_2_tmp +
            c1*med_n_2_to_med_n_1_tmp +
            c4*med_n_1_to_fast_n_4_tmp + c3*fast_n_4_to_fast_n_3_tmp + c2*fast_n_3_to_fast_n_2_tmp +
            c1*fast_n_2_to_fast_n_1_tmp +
            c0*fast_n_1_to_center_tmp +
            c1*center_to_fast_p_1_tmp + c2*fast_p_1_to_fast_p_2_tmp + c3*fast_p_2_to_fast_p_3_tmp +
            c4*fast_p_3_to_fast_p_4_tmp +
            c1*fast_p_4_to_med_p_1_tmp + c2*med_p_1_to_med_p_2_tmp + c3*med_p_2_to_med_p_3_tmp +
            c4*med_p_3_to_med_p_4_tmp +
            c1*med_p_4_to_slow_p_1_tmp + c2*slow_p_1_to_slow_p_2_tmp + c3*slow_p_2_to_slow_p_3_tmp +
            c4*slow_p_3_to_slow_p_4_tmp ;

        //Writing the data to next FIFOs
        slow_n_4_to_slow_n_3[slow_n_4_to_slow_n_3_ptr] = in;
        slow_n_3_to_slow_n_2[slow_n_3_to_slow_n_2_ptr] = slow_n_4_to_slow_n_3_tmp;
        .....
        .....
        .....
        slow_p_2_to_slow_p_3[slow_p_2_to_slow_p_3_ptr] = slow_p_1_to_slow_p_2_tmp;
        slow_p_3_to_slow_p_4[slow_p_3_to_slow_p_4_ptr] = slow_p_2_to_slow_p_3_tmp;

        //Advancing the pointers
        slow_n_4_to_slow_n_3_ptr < SLICE_SIZE - 1 ? (++slow_n_4_to_slow_n_3_ptr) : (slow_n_4_to_slow_n_3_ptr = 0);
        slow_n_3_to_slow_n_2_ptr < SLICE_SIZE - 1 ? (++slow_n_3_to_slow_n_2_ptr) : (slow_n_3_to_slow_n_2_ptr = 0);
        .....
        .....
        .....
        slow_p_2_to_slow_p_3_ptr < SLICE_SIZE - 1 ? (++slow_p_2_to_slow_p_3_ptr) : (slow_p_2_to_slow_p_3_ptr = 0);
        slow_p_3_to_slow_p_4_ptr < SLICE_SIZE - 1 ? (++slow_p_3_to_slow_p_4_ptr) : (slow_p_3_to_slow_p_4_ptr = 0);
    }
}

```

Figure 4.9: Code snippet of the developed single kernel version.

increasing RAM utilization and showing a similar throughput and power consumption and in some cases a worse throughput. This shows that SODA's approach of implementing FIFOs is better than Intel's documented shift registers in implementing stencil kernels. Coarse and fine clustering levels show similar results. Both versions utilize approximately half the number of ALM and register units compared to the original none-clustering level. Both utilized negligibly less RAMs and consumed negligibly less power. Frequency and throughput however, were badly impacted for both versions in the case of 256×256 slice size. The combined FIFOs version shows generally a better resource utilization than the original version. However, it has a higher resource utilization than Coarse and Fine version in ALM, Registers and RAM units. For the throughput and frequency, it shows better results than the original, Coarse and fine versions in the case of 256×256 slice size while it shows worst result in the case of 128×128 slice size. The combined FIFOs and PE version shows better results than all of the previously mentioned versions in terms of resource utilization, frequency, power consumption and throughput. The Full clustering and single kernel versions, however, show significantly better results than all other versions. They both achieve very similar power consumption and throughputs. The single kernel however, utilizes slightly less resources (especially registers and ALMs) due to the elimination of channels. Hence, for $UF = 1$ and $T_s = 1$, throughput improvement of all versions did not exceed 9% and power consumption reduction did not exceed 4%. Even though resource utilization was significantly reduced (by up to 61% for some components), the reduction in RAM utilization reached 33% and 14% for the 128×128 and 256×256 slice sizes, respectively. Still, the reduction was not enough to fit the 512×512 slice.

Using one kernel might allow fitting larger slice sizes in future FPGA generations with more RAMs. This experiment also showed that smaller slices achieve better power efficiencies for various values of UF and Ts.

Next, experiments were run using the one kernel approach with five fused time steps (i.e. $T_s=5$) and two values of UF: 1 and 4 to see the effect on power and throughput again at the two slice sizes of 128×128 and 256×256 . Table 4.12 and Table 4.13 show the comparison of the resource utilization, power, and throughput between the original SODA and the developed one kernel approach for the 128×128 slice size for UF=1 and UF=4, respectively. These results show that the improvement in resource utilization and power efficiency become very significant at higher values of UF and T_s . At UF=4 and $T_s=5$, the single kernel approach reduces adaptive logic modules (ALMs), registers, and BRAM usage by 84%, 79%, and 66%, respectively. This leads to 34% reduction in power consumption at the same throughput as the original SODA. For the 256×256 slice size (Table 4.15 and Table 4.16), in addition to the significant reduction in resource utilization, the throughput increased by ~4% while the power went down by 25% at UF=4 and $T_s=5$. These results are very significant showing that the single or full kernel clustering approaches improve the power efficiency of the FPGA implementation of the stencil from 1.81 W/GB/s to 1.32 W/GB/s. They also show that power efficiency scales linearly with UF and T_s .

Table 4.17 shows the power dissipation components breakdown of the single kernel version for a $128 \times 128 \times 1024$ grid, UF=4 and $T_s=5$. Compared to Table 4.10, contribution of the core circuit consumption to the total power consumption has decreased significantly from 56.4% to 34.1%.

Table 4.11: FPGA resource utilization, attained frequency, power consumption, and throughputs obtained for the different kernel clustering options for a slice size of 128×128 , UF=1, and Ts=1. Maximum Power Efficiency = 16.5 W/GB/s (for the single kernel).

	FPGA Resource Utilization											
	ALM			Registers			DSP			RAM		
Kernel Version	value	ratio	value	value	ratio	value	value	ratio	value	value	ratio	value
Original	25,596	1.00	71,717	1.00	25	1.00	550	1.00	28	1.00	500	1.00
Coarse	11,821	0.46	37,648	0.52	25	1.00	428	0.78	50	1.79	500	1.00
Fine	11,821	0.46	37,648	0.52	25	1.00	428	0.78	50	1.79	500	1.00
Full	9,499	0.37	30,376	0.42	25	1.00	386	0.70	41	1.46	456.67	0.91
Shift Register	25,217	0.99	68,570	0.96	25	1.00	560	1.02	20	0.71	500	1.00
Combined FIFOs	14,812	0.58	45,118	0.63	25	1.00	439	0.80	64	2.29	460	0.92
Combined FIFOs and PE	9,495	0.37	30,142	0.42	26	1.04	370	0.67	28	1.00	500	1.00
Single kernel	8,869	0.35	27,725	0.39	25	1.00	366	0.67	27	0.96	500	1.00

Table 4.12: FPGA resource utilization, attained frequency, power consumption, and throughputs obtained using the single kernel for a slice size of 128×128 , UF=1, and Ts=5. Maximum Power Efficiency = 3.83 W/GB/s (for the Full kernel clustering).

	FPGA Resource Utilization											
	ALM			Registers			DSP			RAM		
Kernel Version	value	ratio	value	value	ratio	value	value	ratio	value	value	ratio	value
Original	107,094	1.00	289,890	1.00	125	1.0	2,338	1.00	100	1.00	470.22	1.00
Full	23,108	0.22	75,836	0.26	125	1.00	1,513	0.65	123	1.23	480.00	1.02
Single kernel	22,023	0.21	73,329	0.25	125	1.0	1,458	0.62	153	1.53	472.22	1.00

Table 4.13: FPGA resource utilization, attained frequency, power consumption, and throughputs obtained using the single kernel for a slice size of 128×128 , UF=4, and Ts=5. Maximum Power Efficiency = 0.99 W/GB/s (for the single kernel).

	FPGA Resource Utilization											
	ALM			Registers			DSP			RAM		
Kernel Version	value	ratio	value	value	ratio	value	value	ratio	value	value	ratio	value
Original	334,702	1.00	910,545	1.00	500	1.00	4,186	1.00	714	1.00	500.00	1.00
Full	58,178	0.17	207,158	0.23	500	1.00	1,581	0.38	764	1.07	500.00	1.00
Single kernel	52,388	0.16	192,763	0.21	500	1.00	1,420	0.34	596	0.83	500.00	1.00

Table 4.14: FPGA resource utilization, attained frequency, power consumption, and throughputs obtained for the different kernel clustering options for a slice size of 256×256 , UF=1, and Ts=1. Maximum Power Efficiency = 17.65 W/GB/s (for the single kernel).

Kernel Version	FPGA Resource Utilization										Frequency (MHz)		Power(W)		Throughput (GB/s)	
	ALM		Registers		DSP		RAM		MLAB		value	ratio	value	ratio	value	ratio
	value	ratio	value	ratio	value	ratio	value	ratio	value	ratio						
Original	27,645	1.00	77,751	1.00	25	1.00	1,318	1.00	28	1.00	456.67	1.00	36.0	1.00	1.83	1.00
Coarse	14,019	0.51	45,427	0.58	25	1.00	1,196	0.91	50	1.79	400.00	0.88	34.7	0.96	1.60	0.87
Fine	14,019	0.51	45,427	0.58	25	1.00	1,196	0.91	50	1.79	400.00	0.88	34.7	0.96	1.60	0.87
Full	11,741	0.42	38,563	0.50	25	1.00	1,154	0.88	43	1.54	483.3	1.06	35.2	0.98	1.93	1.06
Shift Register	27,572	1.00	74,532	0.96	25	1.00	1,328	1.01	20	0.71	429.17	0.94	35.7	0.99	1.72	0.94
Combined FIFOs	16,690	0.60	51,212	0.66	25	1.00	1,208	0.92	67	2.39	460.00	1.01	35.4	0.98	1.84	1.01
Combined FIFOs and PE	11,383	0.41	36,217	0.47	25	1.00	1,140	0.86	38	1.36	462.50	1.01	34.8	0.97	1.85	1.01
Single kernel	10,697	0.39	33,401	0.43	25	1.00	1,138	0.86	51	1.82	500.00	1.09	35.3	0.98	2.00	1.09

Table 4.15: FPGA resource utilization, attained frequency, power consumption, and throughputs obtained using the single kernel for a slice size of 256×256 , UF=1, and Ts=5. Maximum Power Efficiency = 5.06 W/GB/s (for the Full kernel clustering).

Kernel Version	FPGA Resource Utilization										Frequency (MHz)		Power(W)		Throughput (GB/s)	
	ALM		Registers		DSP		RAM		MLAB		value	ratio	value	ratio	value	ratio
	value	ratio	value	ratio	value	ratio	value	ratio	value	ratio						
Original	117,561	1.00	322,069	1.00	125	1.00	6,178	1.00	100	1.00	470.00	1.00	52.4	1.00	9.42	1.00
Full	34,295	0.29	110,538	0.34	125	1.00	5,353	0.87	121	1.21	463.33	0.99	46.9	0.90	9.27	0.98
Single kernel	33,346	0.28	107,802	0.33	125	1.00	5,298	0.86	151	1.51	458.33	0.98	46.6	0.89	9.18	0.97

Table 4.16: FPGA resource utilization, attained frequency, power consumption, and throughputs obtained using the single kernel for a slice size of 256×256 , UF=4, and Ts=5. Maximum Power Efficiency = 1.32 W/GB/s (for both, the single and Full kernel clustering).

Kernel Version	FPGA Resource Utilization										Frequency (MHz)		Power(W)		Throughput (GB/s)	
	ALM		Registers		DSP		RAM		MLAB		value	ratio	value	ratio	value	ratio
	value	ratio	value	ratio	value	ratio	value	ratio	value	ratio						
Original	346,302	1.00	942,573	1.00	500	1.00	8,186	1.00	394	1.00	446.67	1.00	64.7	1.00	35.71	1.00
Full	67,466	0.19	232,395	0.25	500	1.00	5,582	0.68	445	1.13	483.33	1.08	50.9	0.79	38.62	1.08
Single kernel	61,807	0.18	218,271	0.23	500	1.00	5,420	0.66	279	0.71	462.50	1.04	48.7	0.75	36.97	1.04

Table 4.17: Power dissipation components breakdown of single kernel version for a grid size of 128x128x1,024, UF=4, Ts=5.

Total Thermal Power Dissipation = 39,467.22 mW	
Total Transceiver Thermal Power Dissipation = 7,476.53 mW (18.9%)	
Transceiver Standby Thermal Power Dissipation	3,164.15 mW
Transceiver Dynamic Thermal Power Dissipation	4,312.38 mW
Total I/O Thermal Power Dissipation = 11,934.75 mW (30.2%)	
I/O Standby Thermal Power Dissipation	1,787.97 mW
I/O Dynamic Thermal Power Dissipation	10,146.78 mW
Device Static Thermal Power Dissipation = 6,653.19 mW (16.8%)	
Core Dynamic Thermal Power Dissipation = 13,485.72 mW (34.1%)	

Table 4.18: Tile dimensions used to evaluate tiling stage performance.

Parameter	Range of Values
Fast dimension (z-axis)	64, 128, 256, 512
Intermediate dimension (y-axis)	64, 128, 256, 512

4.4 Evaluating the Effect of Tile Size on Tiling Stage Performance

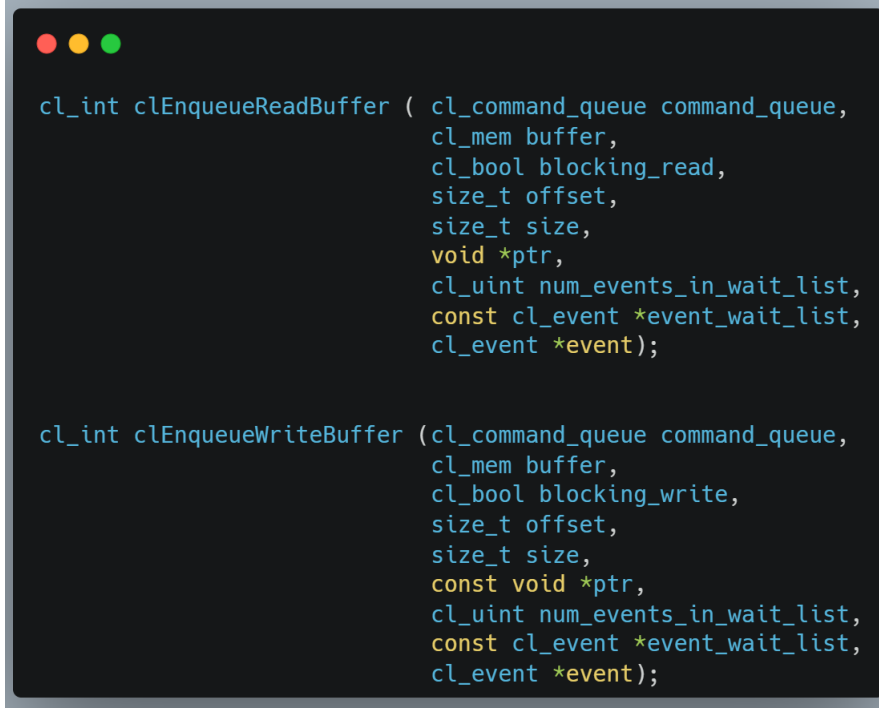
An experiment was designed to evaluate the performance of the tiling stage of the proposed framework as a function of the tile size and shape. Since SODA implementation streams the grid along the 3rd dimension, only the size of the 2-D slice of the grid is tuned. The 3rd dimension is set to 1024 points. Table 4.18 shows values assigned for each grid dimension. All tiles are created from an input grid of size $1024 \times 1024 \times 1024$. Table 4.19 shows the tiling stage throughput for the various combinations of z & y-dimensions. These results show that the throughput increases with the fast dimension but is fairly independent of the intermediate dimension.

Table 4.19: Tiling throughput (GB/s) as a function of fast x (z-axis) and intermediate (y-axis) dimensions.

		Fast dimension (z-axis)			
		64	128	256	512
Intermediate Dimension (y-axis)	64	4.44	4.69	5.06	5.55
	128	4.03	4.69	5.06	5.55
	256	4.04	4.69	5.06	5.56
	512	4.03	4.69	5.06	5.56

4.5 Evaluating the Effect of Tile Size on Sending/Receiving To/From FPGA's Global Memory

Another experiment was conducted to study the effect of the total tile size on the throughput of sending and receiving tiles to/from the FPGA. Once a tile is created (i.e. copied from the grid to another buffer on the host), it is sent to the FPGA as a block of contiguous memory. In this experiment, the block size was varied and the throughput for sending/receiving to/from the FPGA's on-board RAM was measured. Non-blocking OpenCL APIs were used to issue commands of sending and receiving. Figure 4.10 shows the OpenCL code for the sending and receiving functions. Figure 4.11 shows the result of this experiment. Throughputs of sending and receiving follow a similar pattern and both increase with the block size. The maximum throughputs achieved were ~ 8.5 GB/s for sending and ~ 7 GB/s for receiving at the largest possible block size 2^{32} Bits or 4 GB. This is approximately 50% of the maximum theoretical throughput of the 16 lane PCIe gen 3 which is 16 GB/s. At the optimum tile size ($256 \times 256 \times 1024$ or 2^{29} bits or 64 MB), the sending and receiving throughputs



```

cl_int clEnqueueReadBuffer ( cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_read,
                             size_t offset,
                             size_t size,
                             void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event);

cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_write,
                             size_t offset,
                             size_t size,
                             const void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event);

```

Figure 4.10: The sending and receiving functions utilizing OpenCL APIs.

were 7GB/s and 5.5 GB/s respectively. This experiment actually demonstrated the limitations of OpenCL APIs.

4.6 Evaluating the Overall Performance of the Framework

The performance of the developed framework was evaluated using a grid size of $1024 \times 1024 \times 1024$. In this experiment, UF was set to 4 and two parameters were tuned: the tile size and the number of fused time steps (Ts). These parameters have a direct effect on the number of tiles in each time step and the size of the Halo (overlapped) region between tiles which in turn impact the overall latency. The 3rd dimension (slow dimension) of all tiles was fixed at 1024 points for all cases. Table [4.20](#) summarize the

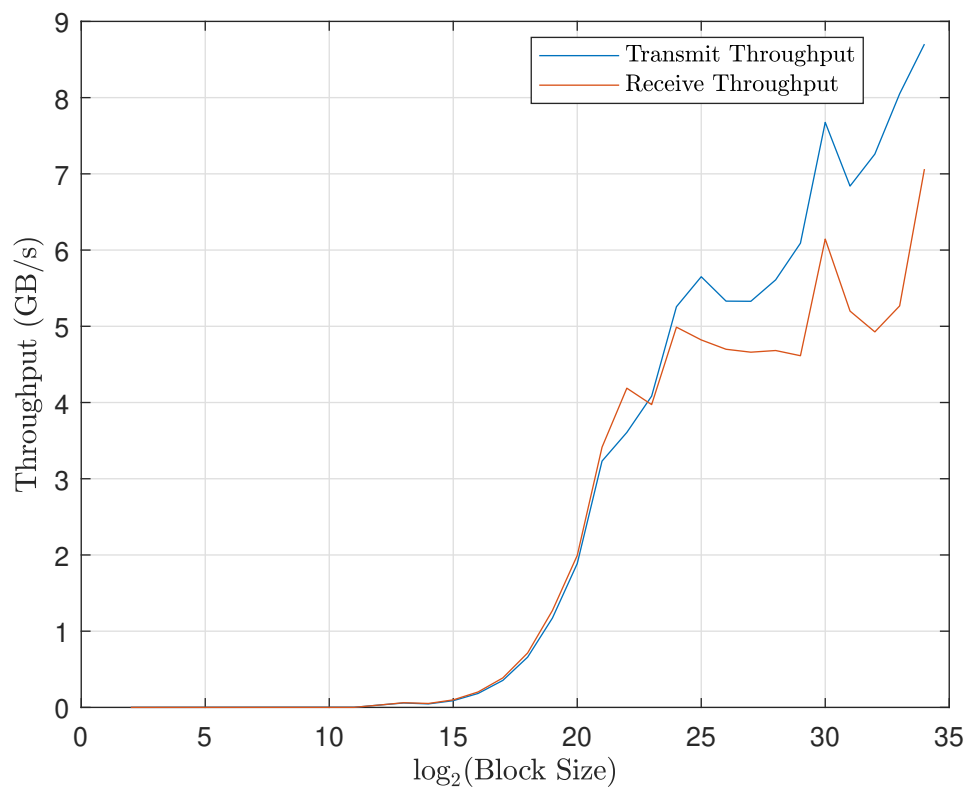


Figure 4.11: Throughput vs. block size.

Table 4.20: Total latency in seconds of computing the whole 3-D grid (from the start of tiling the first tile till the untiling of the last tile) versus tile size and Ts.

Tile Size	Number of Fused Iterations (Ts)				
	1	2	3	4	5
128×128	1.34	1.63	1.62	1.96	2.30
256×256	1.68	1.73	1.71	1.74	1.78
512×512	2.75	2.74	-	-	-

total latency for computing a whole grid as a function of tile size and Ts. It shows that the 256×256 slice size is by far the best as was expected. Table 4.21 shows the obtained latencies for each pipeline stage as well as the peak computing throughput versus Ts. These results show how balanced the pipeline is.

4.7 Comparing Framework Performance to CPU-Based and GPU-Based Implementations

Results of the developed framework implementation were compared to the best CPU results reported in [1] and [2] in terms of Giga Lattice Point Updates Per Seconds (GLPUPs). The hardware used in [1] was a single node of Shaheen II supercomputer with two Intel Haswell Xeon E5-2698v3 sockets each with 16 cores running at 2.3 GHz while in [2] a 64-cores AMD Milan-x CPU with 256 MB L3 Cache and 2.45 GHz speed was used. Figure 18 shows the throughputs of the three implementations. As Figure 4.12 shows, the FPGA implementation gave a relative throughput equivalent to ~ 43 Haswell cores and ~ 13 Milan-x cores.

Framework results were also compared against GPU-based implementation results in [3]. The GPU used was NVIDIA A100 40 GB GPU installed in a commodity server

Table 4.21: Minimum latencies in milliseconds of the asynchronous pipeline stages for three sizes as a function of Ts.

Tile size:128x128						
Ts	Tile	Send	Execute	Receive	Untile	Throughput
1	14.7	7.8	8.9	7.3	8.2	7.5 GB/s
2	12.8	7.8	8.9	7.5	7.0	15.0 GB/s
3	15.1	7.7	8.8	7.4	7.3	22.8 GB/s
4	13.4	7.8	9.1	7.4	5.3	29.5 GB/s
5	14.4	7.6	8.7	7.3	5.5	38.4 GB/s
Tile size:256x256						
Ts	Tile	Send	Execute	Receive	Untile	Throughput
1	37.7	31.4	35.6	30.1	30.3	7.5 GB/s
2	38.8	30.9	35.5	31.1	30.8	15.1 GB/s
3	42.0	31.1	36.1	30.3	31.1	22.3 GB/s
4	43.0	31.0	34.8	29.7	29.3	30.9 GB/s
5	48.4	30.1	38.2	29.4	30.7	35.1 GB/s
Tile size:512x512						
Ts	Tile	Send	Execute	Receive	Untile	Throughput
1	88.9	124.1	151.2	155.2	88.5	7.1 GB/s
2	82.6	119.5	160.6	150.1	86.9	13.4 GB/s
3	-	-	-	-	-	-
4	-	-	-	-	-	-
5	-	-	-	-	-	-

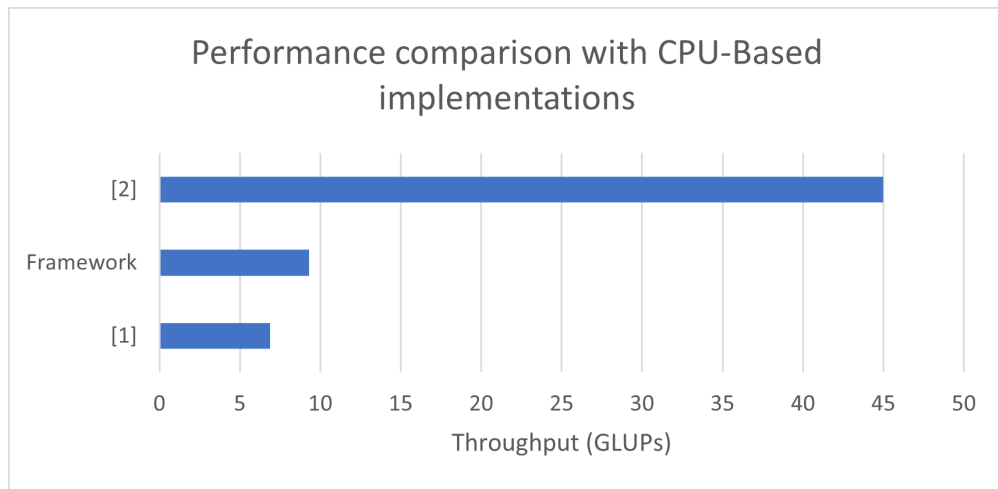


Figure 4.12: Comparison between the framework and both CPU work on [1] and [2].

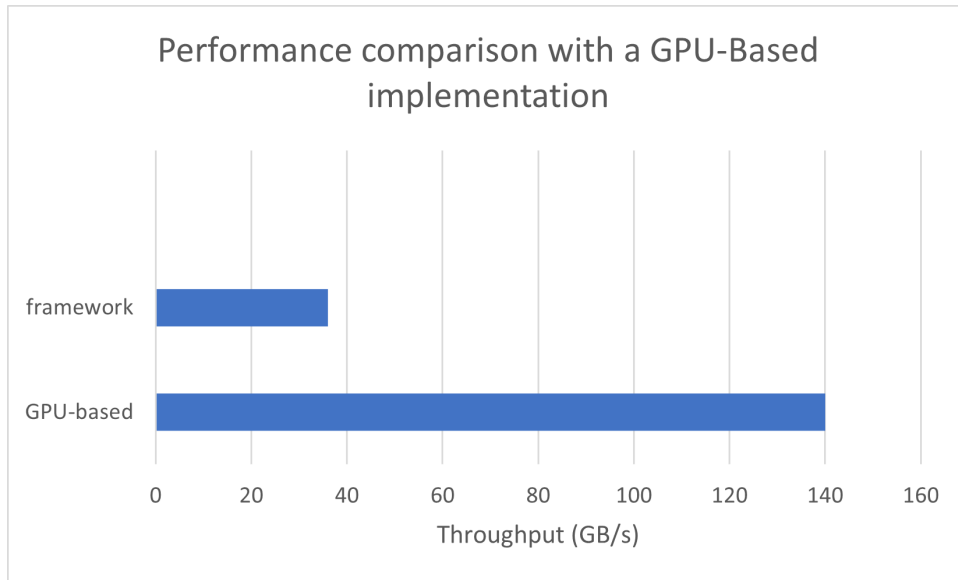


Figure 4.13: Comparison between the framework and GPU-based implementation in [3].

connected over 16 lanes of PCIe Gen 3.0. NVIDIA A100 GPU has 6,912 CUDA cores and 432 Tensor cores. However, only CUDA cores were utilized in the experiment. The video memory of the GPU (VRAM) is connected to the GPU over High-bandwidth Memory 2 (HBM2) with a bandwidth of ~ 1.5 TB/s. The NVIDIA A100 GPU is manufactured using the latest 7 nm lithography technology. Figure 4.13 shows the comparison between the GPU-based implementation and the developed framework in terms of GB/s. The average power consumption reported for the GPU was 261 W. This translates to ~ 1.86 W/GB/s power efficiency compared to the FPGA implementation with ~ 1.32 W/GB/s power efficiency. In fact, the GPU power consumption is equivalent to more than 5 FPGAs. If 5 FPGAs are used (total power = $48.7 \times 5 \sim 244$ W), the aggregated throughput of the 5 FPGAs would be ~ 197 GB/s or $\sim 40\%$ more than the A100 GPU.

4.8 Multi-FPGA Framework Simulation

This experiment aims to study the effect of using multiple FPGA boards on the overall performance of framework. Due to the difficulty of attaching multiple FPGA boards on the host machine, FPGA performance is simulated in this experiment. FPGA related stages (i.e. sending, executing and receiving stages) on the framework are replaced with fixed delays derived from Table 4.21. Two FPGA boards are simulated in this experiment. Input grid size is set to $1024 \times 1024 \times 1024$. Unroll factor (UF) was set to 4 and two parameters were tuned: the tile size and the number of fused time steps (Ts). The 3rd dimension (slow dimension) of all tiles was fixed at 1024 points for all cases. Table 4.22 shows the total latencies of running the simulation for computing a whole 3-D grid. It is observed that latencies decrease sub-linearly with the number of devices. For example, the case of 256x256 tile size and 5 fused time steps shows a decrease in latency of ~70% for 2 FPGA boards. Table 4.23 shows number of tiles per device for 1 and 2 devices as a function of tile size and number of fused time steps (Ts). It is observed for the same grid size and fused time step, the total number of tiles in case of 2 devices is more than the number of tiles in a single device. These additional tiles appears due to the overlapped region between the two sub-grids of both devices. These tiles explain the sub-linear relation between framework latency and number of devices.

Table 4.22: Total latency in seconds of computing the whole 3-D grid versus tile size and Ts for one device and two devices.

Number of devices: 1					
Tile Size	Number of Fused Iterations (Ts)				
	1	2	3	4	5
128×128	1.04	1.13	1.32	1.44	1.86
256×256	1.03	1.04	1.04	1.042	1.06
512×512	1.78	1.79	-	-	-
Number of devices: 2					
Tile Size	Number of Fused Iterations (Ts)				
	1	2	3	4	5
128×128	0.60	0.63	0.82	0.86	1.10
256×256	0.71	0.72	0.74	0.76	0.77
512×512	1.34	1.35	-	-	-

Table 4.23: Number of tiles per device versus tile size and Ts for one device and two devices.

Number of devices: 1					
Tile Size	Number of Fused Iterations (Ts)				
	1	2	3	4	5
128×128	90	100	110	121	156
256×256	25	25	25	25	25
512×512	9	9	9	9	9
Number of devices: 2					
Tile Size	Number of Fused Iterations (Ts)				
	1	2	3	4	5
128×128	45	50	60	66	84
256×256	15	15	15	15	15
512×512	6	6	6	6	6

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

An efficient framework for accelerating the computing of high-order stencils on large 3-D grids on FPGAs was developed, implemented, and evaluated. The developed framework serves two main objective: 1) Allow computing high-order stencils on large 3-D grids that cannot fit on an FPGA board's RAM irrespective of the size of input grid or the number of iterations, 2) Implement a completely asynchronous pipeline that hides the latencies of memory transfers, tiling/untiling the input grid, and bus transfers. The first objective was achieved by tiling the input grid on the host machine while the latter one was achieved by completely overlapping tiling and untiling processes, and memory and bus transfers with the on-FPGA stencil computations without the need for strict synchronization mechanisms that would have resulted in considerable delays. The framework consists of 5 asynchronous fully-pipelined stages: tiling stage, sending stage, computing stage, receive stage and untiling stage. It is

composed of two parts: Stencil kernel and host application. The Stencil kernel was implemented using SODA framework which utilizes optimum on-FPGA data reuse buffers. The framework's host application is a multi-threaded application that was optimized to manage, orchestrate, and balance the execution of the pipeline's stages. Furthermore, the framework was extended to support two FPGA devices attached to the host CPU.

The proposed framework was fully implemented on an Intel Stratix 10 FPGA attached to a dual socket Xeon host CPU. Experimental results show that maximum achievable throughput was 39 GB/s (bounded by the FPGA board's RAM bandwidth). This is equivalent to the throughput of ~43 Haswell cores from Intel or ~13 Milan-x cores from AMD implementing the same stencil on 3-D grids with the same size. The framework also achieved power efficiency of 1.32 W/GB/s. This power efficiency is 40% better than the state-of-the-art A100 GPU from Nvidia.

The following are possible extensions to this work in the future:

- Distributing stencil computation load between the host machine and the FPGA.

This is to take advantage of idle host cores,

- Computing boundary conditions on the host machine. Again, to relieve the FPGA board's RAM and exploit idle cores on the host CPU.

REFERENCES

- [1] V. Etienne, T. Tonellot, T. Malas, H. Ltaief, S. Kortas, P. Thierry, and D. Keyes, “High-performance seismic modeling with finite-difference using spatial and temporal cache blocking,” vol. 2017, no. 1, pp. 1–5, 2017. [Online]. Available: <https://www.earthdoc.org/content/papers/10.3997/2214-4609.201702324>
- [2] L. Qu, H. Ltaief, and D. Keyes, “Designing tomorrow’s algorithms on today’s architectures: A case study with stencil computations using amd 3d v-cachetm technology,” Sep 2022. [Online]. Available: <https://cemse.kaust.edu.sa/ecrc/news/designing-tomorrows-algorithms-todays-architectures-case-study-stencil-computations-using-amd>
- [3] A. Ibrahim and S. A. Muhammad E.S. Elrabaa, “Exploring efficient fpga acceleration of high-order 3-d stencil computation on large data grids.”
- [4] “Opencl overview - the khronos group inc.” [Online]. Available: <https://www.khronos.org/opencl/>
- [5] H. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-based computing systems with openCL*, 11 2017.

- [6] R. Tay, *OpenCL Parallel Programming Development Cookbook*. Packt Publishing, 2013.
- [7] H. R. Zohouri, A. Podobas, and S. Matsuoka, “Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 153–162. [Online]. Available: <https://doi.org/10.1145/3174243.3174248>
- [8] C. Du and Y. Yamaguchi, “High-level synthesis design for stencil computations on fpga with high bandwidth memory,” *Electronics*, vol. 9, no. 8, p. 1275, Aug 2020. [Online]. Available: <http://dx.doi.org/10.3390/electronics9081275>
- [9] H. Ren, “A brief introduction on contemporary high-level synthesis,” in *2014 IEEE International Conference on IC Design & Technology*, 2014, pp. 1–4.
- [10] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza, “A high-level synthesis flow for the implementation of iterative stencil loop algorithms on fpga devices,” in *Proceedings of the 50th annual design automation conference*, 2013, pp. 1–6.
- [11] K. Sano, Y. Hatsuda, and S. Yamamoto, “Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 695–705, 2014.

- [12] H. M. Waidyasooriya and M. Hariyama, “Multi-fpga accelerator architecture for stencil computation exploiting spacial and temporal scalability,” *IEEE Access*, vol. 7, pp. 53 188–53 201, 2019.
- [13] S. Wang and Y. Liang, “A comprehensive framework for synthesizing stencil algorithms on fpgas using opencl model,” in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.
- [14] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, “A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops,” in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.
- [15] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, “Opencl-based fpga-platform for stencil computation and its optimization methodology,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1390–1402, 2017.
- [16] Y. Chi, J. Cong, P. Wei, and P. Zhou, “Soda: Stencil with optimized dataflow architecture,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.
- [17] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio, “On how to accelerate iterative stencil loops: A scalable streaming-based approach,” *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, dec 2015. [Online]. Available: <https://doi.org/10.1145/2842615>

- [18] J. Cong, P. Li, B. Xiao, and P. Zhang, “An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers,” in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.
- [19] K. Sano, Y. Hatsuda, and S. Yamamoto, “Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth,” in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 234–241.
- [20] G. Singh, D. Diamantopoulos, S. Stuijk, C. Hagleitner, and H. Corporaal, “Low precision processing for high order stencil computations,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 19th International Conference, SAMOS 2019, Samos, Greece, July 7–11, 2019, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 403–415. [Online]. Available: https://doi.org/10.1007/978-3-030-27562-4_29
- [21] V. Medeiros, A. Barros, A. Silva-Filho, and M. E. de Lima, *High Performance Implementation of RTM Seismic Modeling on FPGAs: Architecture, Arithmetic and Power Issues*. New York, NY: Springer New York, 2013, pp. 305–334. [Online]. Available: https://doi.org/10.1007/978-1-4614-1791-0_10
- [22] D. L. N. Hettiarachchi, V. S. P. Davuluru, and E. J. Balster, “Integer vs. floating-point processing on modern fpga technology,” in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, pp. 0606–0612.

- [23] K. Sano and S. Yamamoto, “Fpga-based scalable and power-efficient fluid simulation using floating-point dsp blocks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2823–2837, 2017.
- [24] H. R. Zohouri, A. Podobas, and S. Matsuoka, “High-performance high-order stencil computation on fpgas using opencl,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 123–130.

VITAE

- Name: Alhussain Ibrahim Hussain Ibrahim
- Nationality: Yemeni
- Date of Birth: 1996-03-20
- Email: *hussainibrahim6@gmail.com*
- Address: al nuzhah Dist, Taif, Saudi Arabia.
- Publications: Exploring Efficient FPGA Acceleration of High-Order 3-D Stencil Computation on Large Data Grids. (In Progress)