High-Level FPGA Accelerator Design for Structured-Mesh-Based Numerical Solvers

by

Kamalavasan Kamalakkannan

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy in Computer Science

Department of Computer Science

April 2023

Contents

Acknowledgement

On this page, I want to convey my appreciation to the individuals and organizations who have provided assistance and inspiration in various ways during the course of this work. Their contributions have been invaluable, and without their help, this work would not have been possible. I am deeply grateful to them for the knowledge, exposure, and experience I have gained during my PhD.

First of all, I would like to thank my supervisors Dr. Gihan Mudalige and Dr. Suhaib Fahmy. Dr. Gihan offered me this fantastic PhD opportunity, being optimistic and flexible to my interests in research. I am thankful to Dr. Gihan not only for his academic guidance but also for being my go-to person for any kind of assistance, from visa applications to submitting my thesis. I appreciate the time and efforts Dr. Gihan dedicated to helping me develop various skills. Supervisor Dr. Suhaib helped me to dive into HLS-based FPGA acceleration and provided the Hardware and software resources necessary to undertake this research work. I express my gratitude to collaborator, Dr. Istvan Reguly, for providing valuable guidance on how to optimally implement applications on GPUs. I like to extend my thanks to my advisors Dr. Ligang He and Dr. Victor Sanchez for their valuable feedback on my progress during the PhD.

I would like to thank the department of computer science, university of Warwick for offering the funding for PhD and the Department of Engineering, Xilinx and Intel for offering the accelerator devices and software. I would like to thank Jacques Du Toit and Tim Schmielau at NAG UK Ltd for providing the C SLV application. I am thankful to John Shanly, Bogdan Pasca, Yohann Yugen for their support and flexibility during my internship at Intel UK.

It is a pleasure to acknowledge the support and friendship of many lab mates and colleagues including, Dr. Gabriele pergola, Dr. Ali Mohammadi Shanghooshabad, Dr. Arun prabhakar, Dr. Viswash Batra, Suneth Ekanayaka, Zaman Lantra, Archi Powell and Zhihao and Megdad kurmanji. I would like to mention friends, Natheesan, Gowtham Maran, Piradeef and Kokulraj for the valuable discussions, inspiration and support given during this work.

Finally, I am grateful to my parents and siblings for their unwavering support, encouragement and guidance throughout my life. My sister Kalaivarny sparked my interest in science through inspiring and motivational conversations. I want to convey my appreciation to my wife Arsitha for being a supportive and encouraging force in my personal and professional growth.

Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy in Computer Science. I, Kamalavasan Kamalakkannan, declare that this thesis titled, 'High-Level FPGA Accelarator Design for Structured-Mesh-Based Numerical Solvers' has been composed by myself and has not been submitted in any previous application for any degree. I confirm that:

- This work was done wholly or mainly while in candidature for the research degree at this University.
- The work presented (including data generated and data analysis) was carried out by the author.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

Portions of this work have appeared in the following publications :

- Parts of Chapter 3 in[33]:
  Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. 2021. High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers, 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Portland, OR, USA, 2021, pp. 1087-1096.
- Parts of Chapter 4 in[35]:
  Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. 2022. High throughput multidimensional tridiagonal system solvers on FPGAs. In Proceedings of the 36th ACM International Conference on Supercomputing (ICS '22). Association for Computing Machinery, New York, NY, USA, Article 19, 1–12.
- Parts of Chapter 5 in[34]:
  Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. 2022. FPGA Acceleration of Structured-Mesh-Based Explicit and Implicit Numerical Solvers using SYCL. In International Workshop on OpenCL (IWOCL'22). Association for Computing Machinery, New York, NY, USA, Article 19, 1–11.

The tridiagonal solver library and applications developed in this work are available as open-source software :

Git Repositories:

- *StencilsOnFPGA*[37] - `https://github.com/OP-DSL/StencilsOnFPGA`
  Explicit stencil solver-based applications used for benchmarking Xilinx and Intel FPGAs with Nvidia V100 in Chapters 3 and 5
- *Tridsolver-FPGA*[36] - `https://github.com/OP-DSL/Tridsolver-FPGA`
  Batched Trisolver library for Xilinx and Intel FPGAs along with 2D and 3D ADI applications used for benchmarking FPGAs with Nvidia-V100 GPU in Chapters 4 and 5

Abstract

Field Programmable Gate Arrays (FPGAs) have become highly attractive as accelerators due to their low power consumption and re-programmability. However, a key limitation is the time and know-how required to program them. Even with high-level synthesis tools, they still require significant hand-tuned/low-level customizations and design space exploration to gain good performance. The need to program FPGAs using the dataflow programming model, much less well known and practised by the high-performance computing (HPC) community, is a major barrier for adoption for HPC.

The underlying motivation of this work is to bridge this gap - attaining near-optimal performance vs the ease of programming. To this end, we target the important class of applications based on structured meshes, focusing on numerical algorithms based on explicit and implicit techniques. We leverage the main characteristics of the application class, its computation-communication pattern and the hardware features. For explicit schemes, characterized by stencil computations, we unify the state-of-the-art techniques such as vectorization and unrolling with a number of new high-gain optimizations such as creating perfect data reuse data-paths, batching and tiling. A key new feature is their applicability to multiple stencil loops enabling the development of real-world workloads. For implicit schemes, we re-evaluate the characteristics of the tridiagonal system solver algorithms for FPGAs and develop a new high throughput batched multi-dimensional tridiagonal system solver library with orders of magnitude better performance than the state-of-the-art.

New analytic models are developed to support the solvers, elucidating and modelling the critical path of execution and parameterizing the design. This together with the optimal designs and new library lead to a unified design work-flow for synthesis on FPGAs.

The new workflow is used to implement a range of applications, from simple single stencil designs, multiple stencil loops to solvers with real-world utility. They are synthesized on the currently dominant Xilinx and Intel FPGAs. Benchmarking indicate the FPGAs matching or outperforming the best GPU implementations, the current best traditional architecture device solution. Over 30% energy saving can also be observed. The performance model demonstrates over 85% accuracy.

The thesis discusses the determinants for these applications to be amenable for FPGA implementation, providing insights into the feasibility and profitability of a design. Finally we propose initial steps in automating the workflow to be used through a DSL.

Abbreviations

ADI	Alternating Direction Implicit	16
ALM	Adaptive Logic Modules	63
ALU	Arithmetic Logic Unit	7
AoS	Array of Structure	27
API	Application Programming Interface	83
ASIC	Application Specific Integrated Circuit	1
AXI	Advanced eXtensible Interface	27
BTCS	Backward Time Centered Space	127
CFD	Computational Fluid Dynamics	2
CGRA	Coarse Grained Reconfigurable Arrays	8
CPU	Central Processing Unit	6
CUDA	Compute Unified Device Architecture	64
DDR4	Double Data Rate 4	7
DSL	Domain Specific Language	2
DSP	Digital Signal Processing	63
FDM	Finite Difference Method	11
FEM	Finite Element Method	11
FIFO	First Input First Output	23
FPGA	Field Programmable Gate Array	vii
FTCS	Forward Time Centered Space	126
FVM	Finite Volume Method	11
GPU	Graphical Processing Unit	6
HBM	High Bandwidth Memory	32

HDL	Hardware Description Language	1
HLS	High Level Synthesis	1
HPC	High Performance Computing	1
ISL	Iterative Stencil Loop	5
LUT	Look Up Table	23
MISD	Multiple Instruction Multiple Data	65
OPS	Oxford Parallel library for Structured mesh solvers	5
PCIe	Peripheral Component Interconnect Express	10
PCR	Parallel Cyclic Reduction	18
PDE	Partial Differential Equation	6
QoR	Quality of Results	80
RTM	Reverse Time Migration	39
SIMT	Single Instruction Multi Thread	65
SLR	Super Logic Region	7
SLV	Stochastic-Local Volatility	58
SM	Streaming Multi-Processor	65
SoA	Structure of Arrays	27
USM	Unified Shared Memory	73

Symbols

B	Batch size
$Block_{valid}$	Valid mesh point update in s spatial block
$BW_{channel}$	Off-chip memory channel Bandwidth
$Clks_{2D}$	Clock Cycles for 2D stencil computation
$Clks_{3D}$	Clock Cycles for 3D stencil computation
D	Stencil Order
$delay_{2D}$	Total delay in number of clock cycles for 2D application from input to output for single iteration
$delay_{3D}$	Total delay in number of clock cycles for 3D application from input to output for single iteration
$FPGA_{dsp}$	Number of DSP units in FPGA
$FPGA_{mem}$	Available on-chip memory on FPGA
f	Operating frequency of design implemented on FPGA
f_u	PCR inner loop unroll factor
g	Number of Tri-diagonal systems interleaved for thomas solver
G_{dsp}	Number of DSP units required to update single mesh point
l_f, l_b	Arithmetic pipeline latency for forward and backward loops
l_{il}	PCR inner loop pipeline latency
m, n, l	Mesh Dimensions
N	Tri-diagonal system size
N_b	Number of blocks in spike solver
N_{CU}	Number of Compute modules
n_{iter}	Number of Iterations
p	Iterative loop unroll factor
p_{dsp}	Possible iterative loop unroll factor under DSP constraint
p_{mem}	Possible iterative loop unroll factor under onchip memory constraint

t	Tile size in Tiled Thomas solver
V	Vectorization Factor
x, y, z	system sizes in each dimensions, similar to m, n, l

List of Algorithms

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

FPGAs have become highly attractive as accelerator architectures by virtue of their high performance, low power consumption, low latency in processing and re-programmability compared to Application Specific Integrated Circuit (ASIC) counterparts. As a result, FPGAs have gained a foothold in a wider range of application domains such as cyber security [13], databases [58], and deep learning [85]. In recent years, the integration of FPGAs as first-class accelerator platforms has also attracted significant interest in the High Performance Computing (HPC) and scientific computing communities, particularly in the financial computing domain [5]. They have also emerged as a potential accelerator platform for cloud computing [21]. However, a key limitation has been the design effort needed to produce performant accelerators for FPGAs, traditionally implemented through Hardware Description Language (HDL) such as Verilog and VHDL. Implementation of FPGA accelerators using HDL requires expertise in digital system design and optimizations. Moreover, HDL implementations usually require longer design and verification time.

Recent work [94, 59] and commercial FPGA vendors have attempted to address this problem with High Level Synthesis (HLS) tools that can translate programs written in standard high-level languages such as C/C++ or SYCL [73]/OpenCL [55] to a low-level HDL implementation. While this approach has improved the programmability of FPGAs considerably they still require low-level customization and hand-tuning to produce design synthesis with optimum performance. Compared to programming traditional architectures such as CPUs or GPUs, the user has to build the memory hierarchy and data path using the supported high-level language constructs. Such data paths and algorithm transformations must be carefully carried out in order to gain performance on FPGAs as it comes with a fixed set of resources and its operating frequency is significantly lower compared to CPUs and GPUs. As such, getting good performance on FPGAs remains a challenging endeavour.

One solution to this problem is a domain-specific approach, leveraging the key characteristics of applications, their computation and communication patterns or motifs to explore the design space on the target accelerator device. Once the best optimization strategy for the target hardware is identified, it can be used to create templates for similar problems. This technique can be used to create high performance libraries and/or high-level Domain Specific Language (DSL) based frameworks that can generate highly optimized target implementation. This strategy has become an important technique in developing performance-portable massively parallel HPC applications given, the increasing diversity of processor architectures [63, 50, 79, 41].

In this thesis, we apply such an analysis to the domain of structured-mesh-based numerical applications targeting FPGAs. These codes frequently appear as the core motif in solvers for the partial differential equations (PDEs). As such, they are used in applications from a wide range of fields, including Computational Fluid Dynamics (CFD), hydro-dynamics, financial computing, and oil/gas exploration simulations. We focus on numerical methods based on (1) explicit techniques, such as stencil solvers, and (2) implicit techniques, the solution of order dependant algorithms, specifically tridiagonal systems solvers. The key characteristic of the explicit solvers is looping over a "rectangular" multi-dimensional set of mesh points using one or more "stencils" to access data. On the other hand, implicit solver applications utilizing the tridiagonal solvers are characterized by solving linear systems of the equation formed by a set of mesh points along each dimension.

Explicit "Stencil" Solvers: Considerable previous research has developed a range of strategies to synthesize optimized FPGA implementations for explicit stencil solvers [23, 84, 10, 96, 40]. Most recent works utilize the HLS tools, usually compiling OpenCL, and targeting both 2D and 3D stencil applications. They develop a number of standard techniques, ranging from basic methods such as cell-parallel/vectorization, unrolling the iterative loop, to more complex transformations such as spatial/temporal blocking (tiling), in order to best utilize FPGA resources for maximum performance. However, many of the previous work target optimizations specific to an application in isolation without developing a design strategy that can be applied to other stencil codes. While some [84, 83] attempt to generalize accelerator implementations for stencil codes, they only target simpler stencil applications without exploiting higher-gain optimizations. **A key gap in the research is the lack of a unifying design strategy particularly focusing on realistic applications.**

Tridiagonal Solvers: Similarly, previous work on tridiagonal system solvers for FPGAs utilized both low-level hardware description languages [54, 87, 93] as well as high-level synthesis tools [88, 49, 47, 48, 82]. They demonstrated the implementation of the standard tridiagonal system solver algorithms (Thomas, PCR, and Spike), evaluating how to best utilize FPGA resources to maximize performance. However, many of these previous works only develop single system solvers in isolation without a design strategy that can be applied to multiple systems and multi-dimensions in general and do not utilize higher-gain optimizations for real-world applications. **There is a lack of systemic approach to choose the best tridiagonal solver for a given application and how best to implement it on modern FPGAs.**

1.1 Contributions

This thesis makes contributions to advance the state-of-the-art by addressing these open questions and disparities on optimally implementing structured mesh-based numerical applications on FPGAs. The main contributions of this work are as follows:

- **Workflow/implementation template (Chapter 3-5):** We propose an implementation *template*, and an accompanying step-wise optimization strategy for conversion of structured-mesh, explicit, iterative stencil applications (Chapter 3) and tridiagonal solver based implicit applications (Chapter 4) to FPGA accelerators. Given hardware resource constraints, we focus on the features of the application that are amenable for FPGA implementation and optimizations for gaining near-optimal performance. A key method, novel in this work, is the batched execution of multiple independent stencil problems on an FPGA.

- **Tridiagonal solver library (Chapter 4):** We examine the algorithmic trade-offs in developing optimal FPGA designs of multiple multi-dimensional tridiagonal system solves. We propose a design and optimization strategy that optimizes based on problem size, dimensionality, number of systems solved, and data-flow paths required. A key contribution is a new tridiagonal solver library developed with our design space exploration, which can be used in the solution of multi-dimensional applications. The new library demonstrates an order of a magnitude speedup compared to state of art Xilinx Library for larger batches of tridiagonal systems.

- **Analytical models (Chapter 3-5):** The design and analysis are supported through the development of analytic models to predict the performance of designs. These models estimate the key resource consumption and enable rapid exploration of the design space to find the optimal design parameters. Resource models provide estimates for determining the feasibility of implementing a given structured mesh-based application on a given FPGA. The models show over 85% accuracy compared to actual benchmark results.

- **Benchmarking (Chapter 3-5):** Targeting current generation Xilinx FPGAs and Intel FPGAs, we present the design and optimization of three contrasting, representative explicit stencil solvers, and two tridiagonal solver-based implicit applications, comparing a range of alternatives based on resource and performance trade-offs. These applications include both 2D and 3D stencil solvers and multiple stencil loops operating on vector elements and multi-dimensional tridiagonal solvers using FP32 and FP64 arithmetic. The use of High Bandwidth Memory (HBM) available on modern FPGAs, to combine multiple dimension solves and explicit loops, along with batched execution of multiple independent solves is novel for this application class. The runtime, bandwidth, and power/energy performance of the FPGA implementations are compared with highly optimized implementations on a traditional accelerator architecture, a modern Nvidia V100 GPU.

- **Automatic code transformation (Chapter 6):** Finally, we bring together the above strands of work to propose an automatable workflow for FPGA implementation of this class of applications. This work utilizes the frontend of popular DSL framework and presents the steps to identify the optimal design parameters and transform DSL into a FPGA implementation. The key technique behind the transformation steps is design templates, making the transformation simple as well as robust.

1.2 Thesis Overview

This chapter provided the overview of the underlying objectives and motivations, identifies the major gaps in the current state of the art, and highlights the specific contributions made by the research presented in this thesis. The subsequent chapters are structured in the following manner:

Chapter 2 presents background and context to this thesis with details on (1) FPGA accelerator ecosystem and basics on programming FPGAs using high level languages (2) solving partial differential equations using stencil solvers (3) state of the art on accelerating explicit stencil solvers on FPGAs (4) solving partial differential equations using implicit numerical schemes and popular tridiagonal solver algorithms and their trade-offs (5) state of the art on accelerating tridiagonal solvers on FPGAs.

Chapter 3 presents a unified and optimized workflow to synthesize structured mesh based explicit numerical applications on FPGAs. High gain optimisations such as batching and tiling are also detailed along with theoretical performance models that indicate the improvement of run-time on FPGAs. The workflow and models are validated using three representative applications implemented on Xilinx U280 FPGA. Performance on FPGA in terms of runtime, power and bandwidth, is benchmarked against the optimal implementation of the same applications on Nvidia V100.

Chapter 4 presents optimal FPGA design for solving small and medium batched tridiagonal systems as well as optimisation for solving larger tridiagonal systems on FPGAs. The best tridiagonal system algorithm for FPGAs for batched processing is determined based on a theoretical comparison of latency and resource consumption using models. Models and proposed FPGA design is validated on Xilinx U280 FPGA for two non-trivial applications. Again FPGA's performance is benchmarked against the optimal implementation of the same applications on Nvidia V100. Experiments reveal over 30% energy saving for complex financial applications' largest configuration compared to the same application on Nvidia-V100.

Chapter 5 applies the workflow developed in Chapter 3 and Chapter 4 through SYCL programming model on intel FPGAs and compares SYCL language-based kernel designs with C++ for Xilinx Vivado based kernel designs. SYCL-specific optimisation to reduce the kernel call overhead for Iterative Stencil Loops (ISLs) is detailed. Two non-trivial applications were implemented on Intel PAC D5005 using SYCL and the performance of it was again benchmarked against the same applications' performance on Nvidia-V100. It was observed that performance matched with model predicted run-time and optimisation applied on Xilinx FPGAs can equally be applied through SYCL on Intel FPGAs.

Chapter 6 develops the steps to automate the workflow developed in Chapter 3 and Chapter 4. This Chapter leverages popular Domain Specific Language (DSL) for structured mesh-based applications, OPS, and devises the steps to transform applications specified through DSL to FPGA target language. It introduces the Oxford Parallel library for Structured mesh solvers (OPS) framework, presents the part of the OPS application that needs to be accelerated on FPGAs and develops transformation steps to get corresponding FPGA implementation with specified parameters. Finally, it provides the algorithm to identify the optimal design parameters for an FPGA implementation.

Chapter 7 summarizes this thesis and presents the concluding remarks. Subsequent chapters identify potential avenues for future research that may build upon the findings of this thesis.

Chapter 2

Background

This Chapter presents and establishes the background on which we build the contributions of the thesis. We present a comprehensive account of the following: (1) terminology, concepts, challenges and optimization techniques on FPGA-based acceleration (2) popular numerical schemes for solving Partial Differential Equations (PDEs) and analysis of their suitability for parallel architectures (3) state of the art on accelerating structured mesh applications on FPGAs highlighting the key findings and research gaps.

2.1 FPGA Accelerator Device and Eco-System

2.1.1 FPGA Accelerator Device Overview

FPGAs differ from traditional Central Processing Units (CPUs) and Graphical Processing Units (GPUs) as they do not present a fixed, general-purpose architecture to be programmed using the software. A software program is made up of a sequence of instructions that are executed on a fixed CPU or GPU architecture that does not change. In contrast, an FPGA must be configured with an architecture, a specific circuit, that implements the computation of a given task. In comparison with CPUs and GPUs, executing the program on FPGAs entails the creation of a data-flow compute pipeline, passing data through the fixed circuit. Such optimized architecture differs from a general CPU ar-

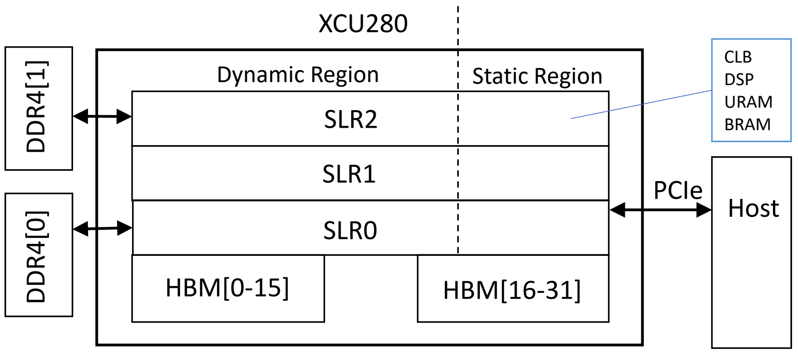


Figure 2.1: Xilinx Alveo U280 FPGA accelerator device.

chitecture as it consists of application-specific caching schemes and multiple Arithmetic Logic Units (ALUs) with dedicated buses and registers. CPUs and GPUs depend on data movement through load and store memory units such as registers, cache and off-chip memory. These operations are costly in terms of latency as well as energy. On the other hand, dedicated data paths on FPGAs and internal data movement not only save the clock cycles but also energy. Additionally, FPGAs operate at a lower clock frequency leading to further energy-efficient execution. The reconfigurability of FPGAs offers a significant advantage over the designing of Application Specific Integrated Circuits (ASICs) which is much more time-consuming and costly and, leads to a fixed architecture that cannot be modified once fabricated.

FPGAs comprise a variety of basic circuit elements to implement a hardware architecture. These circuit elements include ample look-up-tables (LUTs) and registers, large numbers of digital signal processing (DSP) blocks on modern devices, block memories (BRAM/URAM in Xilinx FPGAs and MLAB/M20K in Intel FPGAs), clock modules, and a rich routing fabric to connect these elements into a large logical accelerator architecture. While these resources are primarily suited for the implementation of fixed point integer data paths, they can be used to implement floating point data paths too, though these typically consume significantly more resources for the same computation. Optimizing the datapaths to maximise the achievable frequency, and hence throughput, approaching the limits of what DSP blocks are capable of, is essential in the design of high-performance accelerators on FPGAs [66].

On-chip Random Access Memory (RAM) blocks on FPGAs provide fast and predictable access to data with a fixed latency. These RAM blocks have a combined capacity of typically several tens of megabytes. HLS tools can typically combine multiple such on-chip block memories to form a larger width memory or deeper depth memory. Recent FPGA devices come with coupled High Bandwidth Memory (HBM) [80, 1] that provides a few GBs of capacity and bandwidth in the order of a few hundred GB/s. An FPGA board also include much larger, but slower Double Data Rate 4 (DDR4) memory as external memory. Managing the movement of data between these different types of memory is key to achieving high computational performance. The performance of an FPGA architecture is hard to predict, as it is impacted by various design characteristics beyond the level of the parallelization applied. As a design grows and begins to occupy a larger portion of the FPGA, routing (i.e. connecting all the circuit elements together) becomes more challenging, and routing congestion can reduce the achievable clock frequency and hence overall performance. Careful consideration is required for FPGAs which are partitioned into multiple regions (typically in Xilinx FPGAs, partitioned into Super Logic Regions (SLRs) [90]). This is because the communication links between these regions tend to be slow and can exacerbate routing congestion as the size of the design increases. Therefore, it is crucial to impose placement constraints that restrict the number of connections crossing the partitioned regions.

2.1.2 FPGA Programming

FPGA applications were traditionally developed using HDL like Verilog and VHDL, which require a deep understanding of digital system design to translate algorithms or applications into a parallel digital circuit. This design process is time-consuming and requires extensive functionality verification of implemented circuit. Once the design passed functionality simulations, the HDL implementation will be synthesized into primitive circuit elements such as logic gates and registers, which were then mapped onto available FPGA circuits elements like Look Up Tables (LUTs), registers, block memories, and DSP units. This process was called technology mapping. Next, the technology-mapped elements will be placed in available locations (logic placement) on the FPGA and connected to form the circuit through routing. Additional steps such as post-synthesis and post-implementation simulation, and constraints-based guidance on synthesis, placement, and routing might be necessary to achieve a high-performing circuit implementation. Once routing is complete, a bit stream will be generated from the tool, which could configure the actual FPGA to implement the target circuit. We note here that the time to reconfigure the FPGAs would be in a few seconds while loading kernels in CPUs and GPUs will be in the order of micro-seconds. This whole process (synthesis, placement and routing) will take much longer time compared to compiling a software application.

HDL code generator tools such as MATLAB DSP HDL Toolbox [19], Vision HDL Toolbox [81] and Wireless HDL Toolbox [89] have been developed to ease programming FPGAs by automatically generating HDL implementations while not sacrificing the performance. In these tools, users utilize the optimized pre-built library implementation such as various filter modules for signal processing, to build the required circuit for a given application. Nevertheless, building applications using these tools is more hardware development-oriented than software development. The main drawback is, these tools offer support for specific domains where FPGAs are widely used. Developing applications for new domains would be challenging due to the lack of support for optimized library modules.

Research works have attempted to minimize the time for the application development, FPGA compilation and time to configure the FPGAs through the overlays [72]. Essentially, overlays are configurable architectures that sit on top of the physical layer of FPGAs. It could be Coarse Grained Reconfigurable Arrayss (CGRAs) [31, 7] such as configurable Processing Elements (PEs) or fine-grained Virtual FPGA [2, 46, 8] that implement basic circuit elements on top of physical FPGA. Since the underlying physical circuit won't change when configuring the overlays, it drastically reduces the compilation time to target an application on FPGAs. Configuring overlays is more software-oriented, a custom tool can be used to translate complex applications to overlay configurations. Reconfiguration time for CGRA overlays is much lesser compared to actual physical FPGA reconfiguration times, but it comes at the cost of a performance penalty. Implementing an application through virtual FPGA overlays would require much more resources than implementing that application directly on physical FPGAs. CGRA overlays provide a trade-off between performance and the range of possible configurations. As such, the utilization of overlays on real-world applications is limited.

Previously, the high development overhead of FPGA applications led users to focus on a small set of applications that would provide a better return on the development cost, typically those that could be used in multiple deployments. As a result, the use of FPGAs was limited to certain classes of applications, including embedded, I/O-oriented, and latency-critical applications. However, about a decade ago, major FPGA device vendors such as Xilinx (now acquired by AMD) and Altera (now acquired by Intel) introduced HLS tools. These tools enable users to implement applications using high-level languages such as C/C++/OpenCL/SYCL, with the tools generating functionally matching HDL implementations. In contrast to overlays, generated HDL implementations will be passed to backend tools to synthesize, place and route to get the FPGA configuration bitstream to reconfigure the physical FPGA. In this way, HLS tools minimize the performance penalty at the same time reducing the development time and supports a wider range of domains. Moreover, HLS tools not only reduce the time required for HDL-based designing but also save significant time spent on verification. With HLS tools, users no longer need to be proficient in HDL language, low-level digital design concepts and optimizations, or verification methods to program an FPGA.

While HLS tools are generally effective at generating functionally equivalent HDL implementations, they require guidance and customization of high-level implementation to generate a better-performing implementation. To guide HLS tools, `Pragmas` and `Attributes` are used in C++ for Vivado [25] and SYCL [22]. Some of these transformations, such as loop coalescing, loop fusion, loop interleaving, vectorizations, and global reductions, are also used in CPU optimizations. `Pragmas` and `Attributes` are also used to provide HLS tools with additional information such as dependency distance in loop iterations to avoid pessimistic decisions by the tool. There are optimizations specific to FPGAs, such as exploiting irregular parallelism by pipe-lining the loop and achieving task-level parallelism by data flow optimizations. `Pragmas` directive can be applied to bind the implementation to specific FPGA resources, a variable that can be implemented using the register or on-chip memory. Users can also configure the circuit implementation of a high-level language construct, an array can be partitioned (cyclic, block, or complete) and partitioned blocks could be implemented as individual block memories to obtain higher throughput. Although HLS tools have reduced the required time and effort to obtain an FPGA implementation, applying these customisations and transformations is not a trivial task.

The portion of the program that is accelerated on the FPGA devices is called a kernel. it can be implemented on FPGAs using one of the above FPGA programming methods. To execute a kernel, the FPGA must be reconfigured to load the kernel, necessary data need to be transferred to the device, and the kernel should be started. In order to transfer the data to the device and manage kernel runs, additional hardware modules, such as an external memory controller, Peripheral Component Interconnect Express (PCIe) modules for communication with the host, and modules to manage the board and profile the kernel execution, are required. It is a time-consuming process to implement these support modules on FPGA and establish communication with the host processor, often requiring the operating system to be restarted.

To simplify this process, major FPGA vendors provide a pre-built shell containing necessary hardware modules for data movement, loading the kernel and launching it. During the FPGA board setup process, this shell is loaded onto the assigned FPGA region, called as the static region. The dynamic region is the remaining region where user kernels are loaded. The kernel is loaded into the dynamic region through a process called partial reconfiguration. Partial reconfiguration allows the static region area on FPGA to remain the same and continue to execute while just the dynamic region is reconfigured for loading the new kernel. Since loading kernels using partial reconfiguration is a time-consuming process, it is recommended to load the set kernels and use them repeatedly. The FPGA vendor-specific runtime library communicates with hardware modules in the static region to move the data and execute the kernel.

2.1.3 Loop Latency Estimation

When developing FPGA applications, users often face a tradeoff between performance and the resources/area consumed by the hardware modules. In order to make decisions about the required level of parallelism, it is useful to estimate the performance/latency of the application. One way to estimate the latency of an application implemented using High-Level languages is to analyze the applied parallelism of high-level implementation. Since most part of the application's execution time is spent on executing loops, this section focuses on building a latency model for loops, specifically for `FOR` loops. This model is then utilized in Chapters 3-5 to construct predictive performance models.

```
for(int itr = 0; itr < lB; itr++){
    C[itr] = A[itr] + B[itr];
    E[itr] = C[itr] + D[itr];
}
```

Listing 1: Sequential execution of for loop.

The loop in Listing 1 would be executed sequentially in a naive implementation on an FPGA, with the loop statements executed one after another. Assuming that the integer values are stored in register files (`A`, `B`, `C`, and `D`), instantaneous reading is possible, but writing operations would take a clock cycle. For each iteration, an adder circuit would be required, and it would take two clock cycles to complete the iteration. The number of clock cycles needed to initiate a new loop iteration is called the Initiation Interval (II), which, in this case, is two. The total execution latency of the loop would be $lB \times II$. To improve this latency, the loop could be executed in a pipelined manner by executing two consecutive iterations in parallel, as shown in Listing 2.

```
// clock 0
C[0] = A[0] + B[0]
// clock 1
C[1] = A[1] + B[1]
E[0] = C[0] + D[0]
// clock 2
C[2] = A[2] + B[2]
E[1] = C[1] + D[1]
....
E[lB-1] = C[lB-1] + D[lB-1]
```

Listing 2: Pipelined loop body computation.

Here a new iteration is introduced for each clock, hence $II = 1$. In this case, two adders are required as two addition is done in each clock. This implementation is with a pipeline depth $p_d = 2$, as integer addition can be done in a single clock. Loops with complex computations would require pipeline stages in tens. We note that multiple iterations can't be executed at the same clock as in this case if there is a loop-carried dependency. The total latency in executing this fully pipelined loop will be $lB + 1$. Latency of a pipelined loop (partially/fully) can be generalized to $lB \times II + d_b - 1$.

The latency can be further improved by replicating the pipelined circuits. In this case, the first circuit can execute the 0 to $\lceil lB/2 \rceil - 1$ iterations and the second circuit can execute iterations from $\lceil lB/2 \rceil$ to $lB - 1$. The resultant latency assuming fully pipelined circuits will be $\lceil lB/2 \rceil + 1$. The number of times a circuit is replicated is called as Vectorization factor (V) and latency can be generalized to $\lceil lB/V \rceil \times II + d_b - 1$.

2.2 Structured Mesh-Based Numerical Schemes

The motivating class of applications for this work falls under the domain of structured mesh based numerical algorithms. The performance of such algorithms on parallel architectures depends on their computational and memory access characteristics. In this section, we explore these characteristics for various structured mesh-based numerical methods, using the example of solving the Poisson equation. Researchers and engineers often turn to numerical solutions when analytical solutions are either impossible or require advanced mathematical skills. These methods often involve solving PDEs using a mesh that discretizes the variable's range. The numerical schemes discussed in this section belong to the category of Finite Difference Method (FDM) [43, 44, 75]. Other commonly used structured mesh-based methods include Finite Element Method (FEM) [9] and Finite Volume Method (FVM) [20].

In the FDM, variables are directly discretized and the differential equation is rewritten in terms of discrete differences. A simple differential equation $\frac{\mathrm{d}f(X)}{\mathrm{d}X} = k$ can be numerically solved by discretizing the variable X, where k is a constant. Let's assume variable X is discretized into N values at distance $h = x/N$ as $[x_0, x_1, ..., x_i, ..., x_{N-1}]$. The differential equation at x_i using forward difference can be written as $(f(x_{i+1} - f(x_i)/h =$

k. Similarly, the differential equation can be written as $(f(x_i) - f(x_{i-1})/h = k$ and $(f(x_{i+1} - f(x_{i-1})/2h = k$ using backward and central differences. If the boundary values (function value $f(X)$ at x_0 and x_{N-1}) are known, function values at all the discretized points can be calculated using forward or backward difference.

FDM can also be applied to higher-order partial differential equations involving multiple variables. We develop FDM-based numerical methods to solve 2D Poisson equations which have similarities with the Poisson application we develop later in Chapter 3.

$$\Delta\phi(x, y) = f(x, y) \tag{2.1}$$

$$\frac{\partial^2\phi(x, y)}{\partial x^2} + \frac{\partial^2\phi(x, y)}{\partial y^2} = f(x, y) \tag{2.2}$$

Here the function $f(x, y)$ is known and function $\phi(x, y)$ is unknown. Let's discretize x, y domains and these values are represented by i, j in discretized mesh with resolution h in both coordinates. The second-order central difference-based partial differentiation will result in the following expression

$$\frac{\phi(i+h, j) - 2\times\phi(i, j) + \phi(i-h, j)}{h^2} + \frac{\phi(i, j+h) - 2\times\phi(i, j) + \phi(i, j-h)}{h^2} = f(i, j) \tag{2.3}$$

2.2.1 Explicit Schemes - Stencil Solvers

Explicit numerical scheme forms equations, such that mesh point values can be directly evaluated. One of the popular methods is Jacobi iteration where neighbouring values are used to find an estimate of a mesh point value. As such, Equation 2.3 can be rewritten as Equation 2.4, again this can be specified as an iterative Equation 2.5.

$$\begin{aligned}\phi(i, j) = \frac{1}{4} \times (\ &\phi(i-h, j) \\ &+ \phi(i, j-h) + \phi(i+h, j) + \phi(i, j+h) - h^2 \times f(i, j)\ ) \qquad (2.4)\\ \phi^{N+1}(i, j) = \frac{1}{4} \times (\ &\phi^N(i-h, j) \\ &+ \phi^N(i, j-h) + \phi^N(i+h, j) + \phi^N(i, j+h) - h^2 \times f(i, j)\ ) \qquad (2.5)\end{aligned}$$

Based on boundary conditions and initial values/ previous iteration values, next iteration value of $\phi^{N+1}(i, j)$ in Equation 2.5 can be calculated. As updating $\phi^{N+1}(i, j)$ requires access to $(i, j+h), (i-h, j), (i+h, j), (i, j-h)$ mesh points and it can be specified using the stencil in Figure 2.2.

Mesh point values $\phi(i, j)$ in the time step $N + 1$ can be updated by moving this stencil through this rectangular mesh at time step N and evaluating the expression in equation 2.5. Since there is no dependency on updating any two mesh point values in time step $N + 1$ in this Jacobi iterative method, it is suitable for acceleration on paral-

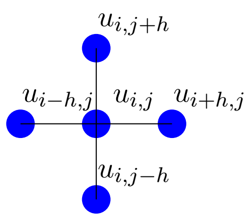


Figure 2.2: Five-point stencil for 2D-Poisson equation.

lel architectures. After iterating through multiple steps, mesh point values will usually converge. Although the explicit numerical scheme equation for this 2D Poisson equation is numerically stable, many explicit scheme equations such as the heat diffusion equation (refer Appendix A) are only stable under certain discretization conditions. Moreover, an explicit scheme requires a larger number of iterations to reach the convergence targeting finer resolutions of discretization compared to the implicit methods that we develop later in this Chapter. Memory requirements for current and next-time steps are proportional to N^2 for a square mesh.

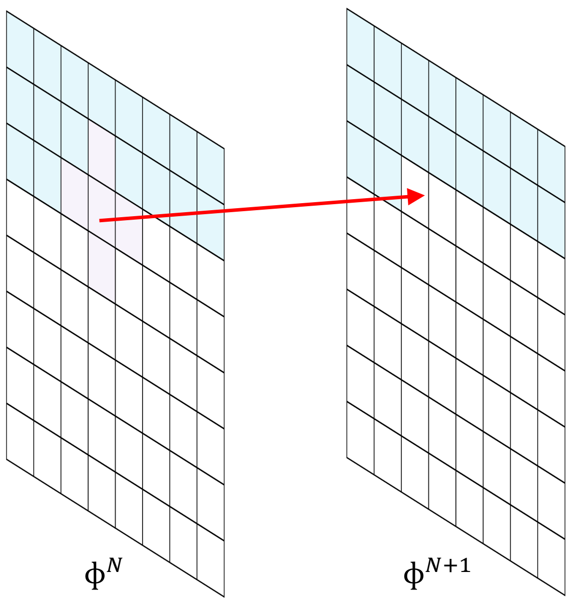


Figure 2.3: Poisson stencil loop updates.

2.2.2 Explicit Schemes - Related Work on FPGAs

Early works [70, 69, 67] targeting FPGAs for stencil computations used Hardware Description Languages (HDL) for describing the architectures. However, the process required extensive hardware knowledge and a time-consuming development cycle. The introduction of High-Level Synthesis (HLS) tools has significantly improved developer productivity and time to design. As such more recent work [84, 83, 62, 10, 40] has utilized HLS tools for implementing FPGA designs for stencil computations. As FPGAs have advanced to incorporate a variety of high bandwidth interfaces and memory types, the system level view of an accelerator architecture has become more important to achieving overall high performance.

The most comprehensive implementation workflow and optimization methodology to date is by Waidyasooriya *et. al* in [84, 83]. The authors use OpenCL and propose an optimization strategy for stencil applications targeting Intel FPGAs. A number of 2-D and 3-D stencil applications are developed through the above strategy, demonstrating up to 950 GFLOPS of achieved computational performance on Intel FPGAs. Runtime and bandwidth performance are compared to conventional GPU and multi-core CPU implementations. The work, however, limits the investigation to applications with only a single stencil loop over the mesh. Multiple stencil loops within a single time-step iterative loop are not considered.

A previous implementation of the 3D Reverse Time Migration (RTM) application, which has similarities to the RTM application we develop later in this work in Chapter 3, can be found in [23]. The implementation uses early-generation Xilinx FPGAs, prior to the introduction of the HLS tool by major vendors, but with designs equivalent to the techniques we use in this work through HLS. Zohouri *et. al* [96] use Intel FPGAs with a design goal to enable unrestricted input sizes for stencil computations without sacrificing performance. They combine spatial and temporal blocking to avoid input size restrictions and employ multiple FPGA-specific optimizations to tackle the added design complexity. The same authors apply these techniques to higher-order stencils in work [95]. The use of spatial and temporal blocking is novel, which Chapter 3 also addresses, but we extend it to variable sized tiling and multi-HBM port implementation, generalizing the technique and incorporating it into our overall design workflow.

A number of previous works have also utilized high-level frameworks for generating efficient FPGA accelerators. The SDSLc framework [62] presents the use of source-to-source translation for generating parallel executables for a range of hardware platforms. These include CPUs, GPUs and FPGAs and details optimizations such as iterative loop unrolling and full data reuse within FPGAs. Similarly, the SODA framework [10] performs several optimizations including perfect data reuse by minimal reuse buffers and data quantization. Additionally, it models the performance and predicts resource consumption, significantly reducing design time. The authors present competitive performance with multi-core CPU implementations and state-of-the-art stencil implementations on FPGAs. The main limitations of the work are fixed tile size and host-based tiling. Due to the DSL's support of only declarative programming, it is not clear whether any limitations exist for porting complex kernels using SODA.

The more recent HeteroCL framework [40] addresses image processing applications. It also supports stencil applications through a SODA back-end as well as through a general back-end. The HeteroCL DSL separates the algorithm from compute, schedules and determines data types, and automatically translates SODA DSL to reflect the iteration unroll factor and other parameters such as data width. A deep single kernel pipeline generated using the above frameworks usually suffers from routing congestion in modern large FPGAs from Xilinx, that incorporate multiple Super Logic Regions (SLRs). This is addressed in work [39] by decoupling the kernel pipeline and assigning placement constraints to limit the inter-SLR crossings. In [17], Dohi *et. al*, use the proprietary MaxCompiler and MaxGenFD high-level design tools to implement finite-difference equations. The work is limited to Maxeler Technologies FPGA platforms and does not compare results with other FPGAs, GPUs or CPUs. The authors of [52] use the polyhedral model and implement a related framework to automatically accelerate iterative stencil loops on a multi-FPGA system. In contrast, [68] develop a scalable streaming Array to implement stencil computations on multiple FPGAs, using a DSL, achieving reduced development time and near-peak performance. Automatic code generation is also used in StencilFlow [15] and has similarities to our design in this work. However, it mainly focuses on non-iterative applications with multiple kernels, hence spanning designs over multiple FPGAs. Batching and spatially blocked optimizations are not attempted.

In contrast to the above works, Chapter 3 of this thesis presents a unifying strategy for the development of FPGA implementations of both 2-D and 3-D stencil applications, including multi-dimensional mesh elements and multiple stencil loops. Chapter 3 of this work incorporate many of the optimization techniques in previous works that have usually been applied in isolation or on a single application. Additionally, Chapter 3 of this work introduce a number of further optimizations such as batching to achieve higher throughput in real-world/production workloads and settings. A predictive analytical model that estimates the feasibility of implementing a given stencil application on a given FPGA platform is also presented. Additionally, the performance of the FPGA accelerators is compared to equivalent highly-optimized implementations of the same applications on modern HPC-grade GPUs, analyzing time to solution, bandwidth, and energy consumption.

2.2.3 Implicit Schemes

Jacobi method-based expression in Equation 2.5 is arranged in a way that $\phi(i,j)$ could be directly evaluated using values in the previous step. In contrast, Equation 2.3 can also, be arranged such that unknown values will form an equation as in 2.6.

$$4 \times \phi(i,j) - \phi(i-h,j) - \phi(i,j-h) - \phi(i+h,j) - \phi(i,j+h) = h^2 \times f(i,j)) \tag{2.6}$$

This equation relates $\phi(i,j)$ with neighbouring elements. Similarly, a set of equations can be formed relating neighbouring elements for all the other mesh points. Such a set of equations will form a matrix as follows for a 4x4 discretization (for $N \times N$ mesh, the coefficient matrix will be the size of $N^2 \times N^2$).

$$
\left(\begin{array}{cccc|cccc|cccc|cccc}
4 & -1 & & & -1 & & & & & & & & & & & \\
-1 & 4 & -1 & & & -1 & & & & & & & & & & \\
& -1 & 4 & -1 & & & -1 & & & & & & & & & \\
& & -1 & 4 & & & & -1 & & & & & & & & \\
\hline
-1 & & & & 4 & -1 & & & -1 & & & & & & & \\
& -1 & & & -1 & 4 & -1 & & & -1 & & & & & & \\
& & -1 & & & -1 & 4 & -1 & & & -1 & & & & & \\
& & & -1 & & & -1 & 4 & & & & -1 & & & & \\
\hline
& & & & -1 & & & & 4 & -1 & & & -1 & & & \\
& & & & & -1 & & & -1 & 4 & -1 & & & -1 & & \\
& & & & & & -1 & & & -1 & 4 & -1 & & & -1 & \\
& & & & & & & -1 & & & -1 & 4 & & & & -1 \\
\hline
& & & & & & & & -1 & & & & 4 & -1 & & \\
& & & & & & & & & -1 & & & -1 & 4 & -1 & \\
& & & & & & & & & & -1 & & & -1 & 4 & -1 \\
& & & & & & & & & & & -1 & & & -1 & 4
\end{array}\right)
\begin{bmatrix}
\phi(0,0) \\ \phi(0,1) \\ \phi(0,2) \\ \phi(0,3) \\ \phi(1,0) \\ \phi(1,1) \\ \phi(1,2) \\ \phi(1,3) \\ \phi(2,0) \\ \phi(2,1) \\ \phi(2,2) \\ \phi(2,3) \\ \phi(3,0) \\ \phi(3,1) \\ \phi(3,2) \\ \phi(3,3)
\end{bmatrix}
=
\begin{bmatrix}
h^2 \times f(0,0) \\ h^2 \times f(0,1) \\ h^2 \times f(0,2) \\ h^2 \times f(0,3) \\ h^2 \times f(1,0) \\ h^2 \times f(1,1) \\ h^2 \times f(1,2) \\ h^2 \times f(1,3) \\ h^2 \times f(2,0) \\ h^2 \times f(2,1) \\ h^2 \times f(2,2) \\ h^2 \times f(2,3) \\ h^2 \times f(3,0) \\ h^2 \times f(3,1) \\ h^2 \times f(3,2) \\ h^2 \times f(3,3)
\end{bmatrix}
\tag{2.7}
$$

Solving the PDEs numerically using this method falls under the category of implicit numerical schemes, as values can't be explicitly evaluated. The benefit is, it is a direct method not requiring multiple iterations for the convergence as in explicit methods and it is unconditionally stable. The drawback of this method is, finding solutions to such a linear system with a high diagonal bandwidth coefficient matrix, requiring sophisticated algorithms. Accelerating such an algorithm in parallel architectures is quite a challenging problem due to data dependency. Moreover, the memory requirement will also be proportional N^4 for a square mesh with size $N \times N$ compared to N^2 for the Jacobi iteration we have seen previously.

2.2.3.1 Alternating Direction Implicit method

As a trade-off, the popular Alternating Direction Implicit (ADI) [18] method provides faster convergence than the explicit methods at the same time offers a better degree of parallelism compared to the above fully implicit method. For the Poisson equation in 2.3, the ADI scheme adds another step $N + \frac{1}{2}$ between N and $N + 1$ for a partial differential equation involving two variables. Equation 2.8 can be formed by using step $N + \frac{1}{2}$ to find the derivative of $\frac{\partial^2 \phi(x,y)}{\partial x^2}$ and step N for the rest in Equation 2.2. Similarly, step $N + \frac{1}{2}$ and $N + 1$ can be used form the Equation 2.9. Those two equations can be reduced to Equations 2.8, 2.9 respectively. A set of equations as in Equations 2.8 along each `x-dim` will form a linear system with a tridiagonal coefficient matrix. Hence there is a need to solve a number of tridiagonal systems when finding the mesh values in step $N + \frac{1}{2}$. This makes it suitable for parallel computing architectures as tridiagonal systems can be solved independently. Many scientific applications are based on the ADI scheme as optimized

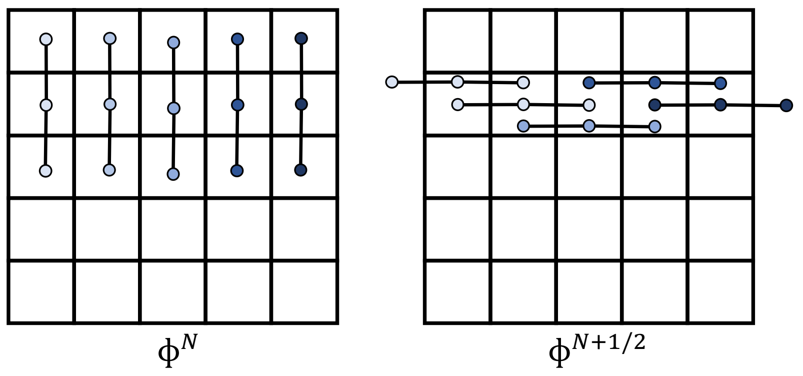


Figure 2.4: Required mesh points for `xsolve` in step N and $N + \frac{1}{2}$.

libraries for solving tridiagonal systems available for in CPUs and GPUs.

$$\begin{aligned} &\phi^{N+1/2}(i+h,j) - 2 \times \phi^{N+1/2}(i,j)\phi^{N+1/2}(i-h,j) \\ &\qquad + \phi^N(i,j+h) - 2 \times \phi^N(i,j) + \phi^N(i,j-h) &= f(i,j) \times h^2 \end{aligned} \tag{2.8}$$

$$\begin{aligned} &\phi^{N+1/2}(i+h,j) - 2 \times \phi^{N+1/2}(i,j) + \phi^{N+1/2}(i-h,j) \\ &\qquad + \phi^{N+1}(i,j+h) - 2 \times \phi^{N+1}(i,j) + \phi^{N+1}(i,j-h) &= f(i,j) \times h^2 \end{aligned} \tag{2.9}$$

$$\begin{aligned} &\phi^{N+1/2}(i+h,j) - 2 \times \phi^{N+1/2}(i,j) + \phi^{N+1/2}(i-h,j) \\ &\qquad = f(i,j) - (\phi^N(i,j+h) - 2 \times \phi^N(i,j) + \phi^N(i,j-h)) \end{aligned} \tag{2.10}$$

$$\begin{aligned} \phi^{N+1}(i,j+h) \quad &- 2 \times \phi^{N+1}(i,j) + \phi^{N+1}(i,j-h) \\ &= f(i,j) - (\phi^{N+1/2}(i+h,j) - 2 \times \phi^{N+1/2}(i,j) + \phi^{N+1/2}(i-h,j)) \end{aligned} \tag{2.11}$$

Equation 2.10 involves three unknowns, namely $\phi^{N+1/2}(i+h,j)$, $\phi^{N+1/2}(i,j)$, and $\phi^{N+1/2}(i-h,j)$, at time step $N + 1/2$, while other values are known at time step N. A system of equations can be constructed using the mesh points shown in Figure 2.4, which results in a tridiagonal matrix. To compute the right-hand side (RHS) values, a 3-point stencil is applied over the mesh at time step N. The mesh point values at time step $N + 1/2$ can be obtained by solving the tridiagonal matrix system in the form of $Au = d$.

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, ..., N-1 \tag{2.12}$$

$$\begin{bmatrix} b_0 & c_0 & 0 & \dots & 0 \\ a_1 & b_1 & c_1 & \dots & 0 \\ 0 & a_2 & b_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N-1} \end{bmatrix} \tag{2.13}$$

The coefficient matrix A in Equation 2.13 has a size of $N \times N$, which is much smaller than the previously encountered matrix size of $N^2 \times N^2$ in Equation 2.7. In order to solve the Poisson equation along both the `X-Dim` and `Y-Dim`, a set of N such matrix systems need to be solved. Equation 2.11 can also be solved by using a set of equations along the `Y-Dim` and utilizing the updated values obtained from Equation 2.10 at time step $N + 1/2$. The ADI method is an iterative solver that solves the system of tridiagonal equations in each direction or variable cyclically, hence the name Alternating Direction Implicit (ADI) method. This approach provides some level of parallelism compared to direct implicit methods, as many smaller matrix systems needs to be solved in the ADI method. Additionally, the ADI method is a more stable numerical method than the explicit stencil-based solvers we have previously examined.

2.2.3.2 Tridiagonal solver algorithms

The solution to tridiagonal systems of equations is well known. The Thomas algorithm [76] (see Algo. 1) carries out a specialized form of Gaussian elimination (assuming non-zero b_i). After the execution of forward loop in Algorithm 1 system will be modified to upper diagonal form as in Equation 2.14. Solution to Thomas algorithm provides the least computationally expensive solution, but suffers from a loop carried dependency. It has a time complexity of $\mathcal{O}(N)$.

$$\begin{bmatrix} 1 & c_0^* & 0 & \dots & 0 \\ 0 & 1 & c_1^* & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} d_0^* \\ d_1^* \\ d_2^* \\ \vdots \\ d_{N-1}^* \end{bmatrix} \tag{2.14}$$

In contrast, the Parallel Cyclic Reduction (PCR) algorithm [24](see Algo. 2), operates on a normalized matrix so that $b_i = 1$ and then for each matrix row i, subtracts multiples of rows $i \pm 2^0, 2^1, 2^2, ..., 2^{P-1}$, where P is the smallest integer such that $2^P \geq N$. This leads to each iteration reducing each of the current systems into two systems of half the size as in Figure 2.5. After P steps, all of the modified a and c coefficients are zero, leaving values for the unknowns u_i. In PCR, the iterations of the inner loop do not depend on each other, which is well suited for traditional multi-core/many-core architectures such as CPUs and GPUs allowing multiple threads to be used to solve each tridiagonal system. However, PCR has a complexity of $\mathcal{O}(N \log N)$ and is more computationally expensive than the Thomas algorithm, which for an FPGA implementation poses an important consideration, (as examined in Section 4.1) due to the limited availability of resources.

Algorithm 1: `thomas`(a, b, c, d, u)

```
1:  $d_0^* \leftarrow d_0/b_0$
2:  $c_0^* \leftarrow c_0/b_0$
3:  for $i = 1, 2, ..., N-1$ do
4:      $r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
5:      $d_i^* \leftarrow r(d_i - a_i d_{i-1}^*)$
6:      $c_i^* \leftarrow r c_i$
7:  end for
8:  $u_{N-1} \leftarrow d_{N-1}$
9:  for $i = N-2, ..., 1, 0$ do
10:     $u_i \leftarrow d_i^* - c_i^* u_{i+1}$
11: end for
12: return $u$
```

Algorithm 2: `pcr`(a, b, c, d, u)

```
1:  for $p = 1, 2, ..., P$ do
2:      $s \leftarrow 2^{p-1}$
3:      for $i = 0, 1, ..., N-1$ do
4:          $r \leftarrow 1/(1 - a_i^{(p-1)} c_{i-s}^{(p-1)} - c_i^{(p-1)} a_{i+s}^{(p-1)})$
5:          $a_i^{(p)} \leftarrow -r(a_i^{(p-1)} a_{i-s}^{(p-1)})$
6:          $c_i^{(p)} \leftarrow -r(c_i^{(p-1)} c_{i+s}^{(p-1)})$
7:          $d_i^{(p)} \leftarrow r(d_i^{(p-1)} - a_i^{(p-1)} d_{i-s}^{(p-1)} - c_i^{(p-1)} d_{i+s}^{(p-1)})$
8:      end for
9:  end for
10: $u \leftarrow d^{(P)}$
11: return $u$
```

$$\begin{bmatrix} 1 & c_0 & & & & & & \\ a_1 & 1 & c_1 & & & & & \\ & a_2 & 1 & c_2 & & & & \\ & & a_3 & 1 & c_3 & & & \\ & & & a_4 & 1 & c_4 & & \\ & & & & a_5 & 1 & c_5 & \\ & & & & & a_6 & 1 & c_6 \\ & & & & & & a_7 & 1 \end{bmatrix} \rightarrow \left[\begin{array}{cccc|cccc} 1 & c_0^* & & & & & & \\ a_2^* & 1 & c_2^* & & & & & \\ & a_4^* & 1 & c_4^* & & & & \\ & & a_6^* & 1 & 0 & & & \\ \hline & & & 0 & 1 & c_1^* & & \\ & & & & a_3^* & 1 & c_3^* & \\ & & & & & a_5^* & 1 & c_5^* \\ & & & & & & a_7^* & 1 \end{array}\right]$$

Figure 2.5: After one iteration of PCR outer loop.

The SPIKE algorithm [60] decomposes the A matrix, into p partitions of size m to obtain the factorization of $A = DS$ where D is a main diagonal block matrix consisting of tridiagonal matrices $A_1, ..., A_p$ and S is the so called spike matrix. The solution to the system then becomes, $DSx = d$ where the system $DY = d$ can be used to obtain Y, and $Sx = Y$ to obtain x. Since matrix D is a simple collection of A_i, each $A_iY_i = d_i$ can be solved independently. Solving $Sx = Y$ requires only solving a reduced penta-diagonal system (see Wang *et al.* [86] for a detailed description). The algorithm therefore operates in three steps: factorization, reduced system solve, and back substitution, where the factorization (LU and UL) has a complexity of $\mathcal{O}(N)$. The reduced system can be solved directly or indeed can be further reduced to a block diagonal system using the truncated-SPIKE variation that ignores the outer diagonals when A is diagonally dominant. The SPIKE algorithm is particularly well suited for solving very large systems on traditional architectures.

2.2.4 Implicit Scheme - Related work on FPGAs

Earlier work implementing tridiagonal system solvers on FPGAs such as by Oliveira *et al.* [54], Warne *et al.* [87] and Zhang *et al.* [93] used low-level Hardware Description languages (HDL) such as VHDL or Verilog for implementing the Thomas algorithm. These designs were restricted to solving 1D or 1D batched tridiagonal systems, instead of full multi-dimensional applications. Oliveira *et al.* [54] pipelined both the forward and backward loops and applied data flow between them and demonstrated the implementation for a smaller 16^3 mesh based application using only on-chip memory.

With the introduction of High-Level synthesis (HLS) tools, a number of more recent works [88, 49, 47, 48] implemented the Thomas, PCR, and Spike algorithms on FPGA using HLS tools. Many of these did not demonstrate the solver working on full applications, with the exception of László *et al.* in 2015 [47] which compared a one factor Black-Scholes option pricing equation using explicit and implicit methods on different architectures such as multi core CPUs, GPUs, and FPGAs. Their implementation, based on the Thomas algorithm, targets a Xilinx Virtex 7 FPGA and effectively pipelines both forward and backward loops but was not able to apply data flow between these two steps and results showed an Nvidia K40 GPU significantly outperforming the FPGA.

Macintosh, *et al.* [49] used OpenCL targeting an Altera Stratix V FPGA to implement the PCR and SPIKE algorithms, showing comparable performance to an Nvidia Quadro 4000 GPU, not including reconfiguration time for the spike kernels. Later, Macintosh, *et al.* [48] used OpenCL to develop `oclspkt`, a library that implements tridiagonal systems solvers targeting FPGAs, GPUs, and CPUs. `oclspkt` uses the truncated spike algorithm for diagonally dominant tridiagonal matrices, and as such does not give exact solutions. Their results show `oclspkt` on an Altera Arria 10GX FPGA performing marginally slower than an Nvidia Quadro M4000 GPU but providing better energy efficiency. The Xilinx library also implements a Douglas ADI solver [18] a multi-dimensional solver based on their PCR based solver [82].

In comparison, the HLS-based synthesis presented in Chapter 4 of this thesis, targets the solution of multiple tridiagonal systems and in multiple dimensions as commonly found in real-world applications. It uses the Thomas algorithm demonstrating that together with techniques such as batching of systems [33], high throughput for small and medium sized systems can be achieved. The Thomas algorithm uses fewer resources than the more computationally intensive PCR algorithm. For larger systems that do not directly fit in a single FPGA, novel Thomas-Thomas and Thomas-PCR solvers are developed in Chapter 4 to handle a number of partitioned systems and then a reduced system solve to exploit the limited available on-chip memory resources of a single FPGA.

Several recent works have also exploited HBM in modern FPGAs [71, 27, 38] showing performance gains and energy savings for memory bandwidth bound applications compared to traditional architectures and FPGA devices without HBM. Multi-dimensional tridiagonal solvers are also bandwidth bound, but, to our knowledge, no previous work has explored the use of HBM capable FPGAs to accelerate them, as this thesis have done through the use of parallel compute units. To our knowledge, the 2D/3D ADI and SLV applications developed in this work, motivated by real-world implicit problems on FPGAs are also novel; SLV being one of the few non-trivial applications using multi-dimensional tridiagonal solvers presented in the literature. The Thomas based solver developed in this paper gives higher performance than the current PCR based Xilinx library, as shown in Section 4.2. Additionally, the analytical performance model and the comparison with a state-of-the-art GPU based tridiagonal solver library gives a much needed frame of reference for evaluating our FPGA design's performance, providing insights into the feasibility and profitability of an FPGA design for realistic workloads.

Chapter 3

Explicit Solvers on FPGAs

Given the appealing features of explicit numerical schemes for parallel implementation, several previous works have attempted to utilize FPGAs for explicit stencil solvers. Early works [70, 69, 67] implemented stencil applications using HDL, some directly implementing stencil applications using HDL and a few works developed specialized processing architecture [67] and utilized it through a compiler. The main limitation being only simple stencil loops were explored. Recent works [84, 83, 62, 10, 40, 96] utilized High-level languages to program FPGAs and attempted optimization such as Vectorization [84], unrolling iterative [84] loops and even spatial and temporal blocking [96] to support larger meshes. The key limitation of the current research works is the lack of a unified methodology and systematic approach for implementing this class of applications on FPGAs.

In contrast to above works, we present a workflow for synthesizing near-optimal FPGA implementations of structured-mesh-based stencil applications for explicit solvers. It leverages key characteristics of the application class and its computation-communication pattern and the architectural capabilities of the FPGA to accelerate solvers for high-performance computing applications. Key new features of the workflow are (1) the unification of standard state-of-the-art techniques with a number of high-gain optimizations such as batching and spatial blocking/tiling, motivated by increasing throughput for real-world workloads and (2) the development and use of a predictive analytical model to explore the design space, and obtain resource and performance estimates. Three representative applications are implemented using the design workflow on a Xilinx Alveo U280 FPGA, demonstrating near-optimal performance and over 85% predictive model accuracy. These are compared with equivalent highly-optimized implementations of the same applications on modern HPC-grade GPUs (Nvidia V100), analyzing time to solution, bandwidth, and energy consumption. Performance results indicate comparable runtimes with the V100 GPU, with over 2× energy savings for the largest non-trivial application on the FPGA. Our investigation shows the challenges of achieving high performance on current generation FPGAs compared to traditional architectures. We discuss determinants for a given stencil code to be amenable to FPGA implementation, providing insights into the feasibility and profitability of a design and its resulting performance.

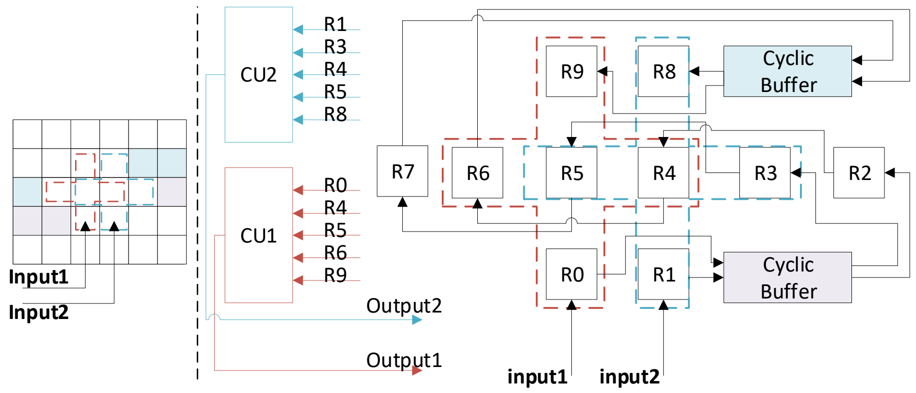


Figure 3.1: Window buffer and factor of 2 vectorizations.

3.1 Accelerator Design for Stencil Computation

To achieve high computational throughput on FPGAs, a custom architecture is designed, which is then implemented using low-level circuit elements such as Look Up Tables (LUTs) and Registers. A data-flow arrangement seeks to map a complex computation to a series of data paths that implement the required computational steps with the movement of data through direct connections. Compared to fixed CPU and GPU architectures where the steps in an algorithm are computed sequentially with intermediate results stored in registers, FPGA compute pipelines can be much deeper and more irregular parallelism can be exploited. Performing a stencil computation will then involve, starting up the pipeline (requiring some clock cycles equal to the pipeline depth) and outputting the result from the computation for each mesh point per clock cycle as a pipelined execution.

For CPU/GPU architectures such a computation is implemented using nested loops, iterating over the mesh and over the neighborhood points. On FPGAs these multiple levels of loops can be unrolled. Retaining an outer loop can be costly due to the need to flush the unrolled inner loop pipeline which can be long. Hence, multi-dimensional nested loops should be flattened to a 1D loop either manually or by using HLS directives such as `loop_flatten`. We have observed that manual flattening still provides the best performance and optimized resource utilization, as current Xilinx HLS compilers can make pessimistic scheduling decisions.

A key approach to gaining the best performance from the above computational pipeline is streaming data from/to external and near-chip memories to/from on-chip block memories to feed the computational pipelines efficiently. A perfect data reuse path can be created by (1) using a First Input First Output (FIFO) buffer to fetch data from DDR4/HBM memory without interruption (allowing burst transfers) to on-chip memory, and then (2) by caching mesh points using the multiple levels of memory, from registers to block memories. Fig. 3.1 illustrates such a data path for a 2D, 2nd order stencil. This technique has previously been referred to as window buffers [23]. A 2D, D order stencil requires D rows to be buffered to achieve perfect data reuse. Similarly, D planes should be buffered for a 3D stencil. The total number of mesh elements needed to be buffered is the maximum

```
for(int itr = 0; itr < DimX*DimY; itr++){
 int i = itr / DimX;
 int j = itr % DimX;
 if(i>0 && j>0 && i<DimX-1 && j<DimY-1){
  out[itr] = (in[itr-1]+in[itr+1] +
        in[itr-DimX]+in[itr+DimX)*0.125 +
        in[itr]*0.5;
  }
}
```

Listing 3: A 2D flattened stencil loop.

number of mesh elements between any two stencil points. BRAM/URAMs can be used to design window buffers by using cyclic buffering. Given their high capacity, URAMs are preferred if the number of elements to be buffered is large.

3.1.1 Stencil Loop Transformation

Implementation of the above-described window buffer for full data reuse in stencil computation requires calculation of individual buffer size, implementing each buffer using on-chip memory and chaining them to form a full window buffers setup. Following steps transform the stencil loop into data flow loop on FPGA using window buffers. Stencil loop in Listing 3 is used for illustrating each step.

- **Loop flattening:** If the original loop is nested, it needs to be flattened, and the data type of the iteration variable should be chosen carefully to accommodate the larger iteration space of the resulting 1D loop. To achieve this, FPGA tools offer template classes for arbitrary-width integer types.
- **Distance calculation:** All memory access indices should be based on the iterative variable (`itr`) and should be ordered according to their index offset with respect to `itr`. For example, in this case, the indices will be {-DimX, -1, 0, 1, +DimX}.
- **Element count:** As the memory accesses will be substituted with a dataflow pipeline, buffering of several elements will be necessary between the memory access points. The number of elements to be buffered between two access points is determined by the gap between the corresponding access points. In this case, it is {DimX-1, 1, 1, DimX-1}.
- **Window buffers**: After determining the necessary number of elements to buffer to be buffered, an array can be used to implement it, with reads and writes occurring at the specified distance. For buffering just a single element, a simple variable assignment or data movement between two registers will suffice. These window buffers can be linked together, as shown in Listing 4.
- **Updating loop iteration bound:** The term ”prime time” refers to the number of iterations needed to generate the first valid output. To determine prime time, we

calculate the number of iterations required to fill the buffers. It equals the number of elements between the stencil update point and the farthest point, assuming the values outside the mesh are zero. In this example, prime time is `DimX`, and we need to increase the number of iterations in the stencil loop by this amount. However, as the loop iteration has now surpassed the mesh's boundary, we need to use conditional statements to prevent out-of-bounds access while accessing global memory.

After following the aforementioned steps, the stencil loop illustrated in Listing 3 can be converted into a flattened loop that only read and write once in each iteration, as shown in Listing 4. This flattened loop can be efficiently pipelined by High-Level Synthesis (HLS) tools, and full data reuse can be attained. Additionally, global memory access within the loop can be replaced with input/output data from a stream interface. This separation of computation and memory access simplifies memory access optimizations for FPGA HLS tools.

```
int pItr = DimX; // prime iterations
float window_1[DimX], window_2[DimX]; // buffer declaration
float s_1_0, s_0_1, s_1_1, s_2_1, s_1_2; // stencil points
for(int itr = 0; itr < DimX*DimY+pItr; itr++){
 int i = (itr-pItr) / DimX;
 int j = (itr-pItr) % DimX;
 int l = itr % (DimX-1); //cyclic index for window buffer

 // transformed memory access
 s_1_0 = window_2[l];
 s_0_1 = s_1_1;
 window_2[l] = s_0_1;
 s_1_1 = s_2_1;
 s_2_1 = window_1[l];
 if(itr < DimX*DimY){ //guard
  s_1_2 = in[itr];
 }
 window_1[l] = s_1_2;

 float res = (s_1_0+s_0_1+s_2_1+s_1_2)*0.125 + s_1_1*0.5;
 if(i>0 && j>0 && i<DimX-1 && j<DimY-1){ // guards
    out[itr-DimX] = res;
  }
}
```

Listing 4: Transformed stencil loop.

3.1.2 Vectorization and Unrolling the Iterative Loop

The design presented above updates only single mesh-point each clock or in each loop iteration. Since there is no dependency between updating two mesh points, Multiple pipelines for the same computation (i.e. loop body or kernel) can be created using HLS directives. This technique, called the cell-parallel method in [84] allows computation of

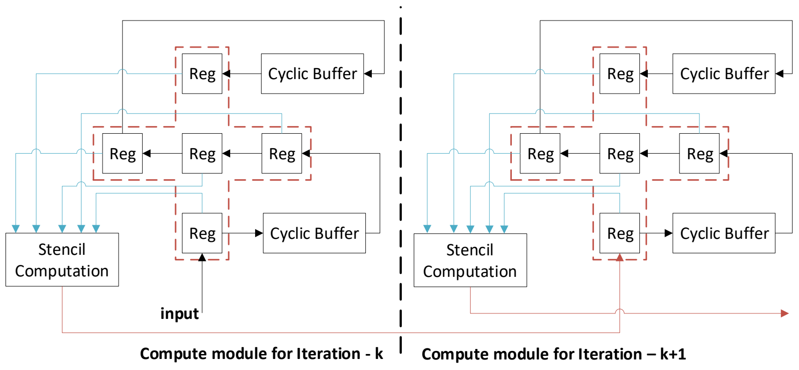


Figure 3.2: Unrolling the iterative loop.

the stencil on multiple mesh points simultaneously. The cell-parallel method is similar to SIMD vectorization on CPUs and SIMT on GPUs but on an FPGA it essentially creates parallel replicas of the computational units as opposed to single vector operations. However the resource availability in an FPGA limits the number of parallel units that can be synthesized on a given device. Figure 3.1 illustrates a factor of 2 implementation, where the vectorization factor represents the number of mesh points updated in parallel.

Another approach that can increase performance is to unroll the iterative loop, which encompasses one or more stencil loops over the rectangular mesh. This allows the results from a previous iteration to be fed to the next iteration without writing back to external (DDR4 or HBM) memory. This scheme, called the step-parallel technique in previous work [84] is illustrated in Figure 3.2. Note how the unrolling yields two "compute modules" in this case. The technique leads to increased throughput without the need for additional external memory bandwidth. However, the unrolling factor depends once more on available FPGA resources and internal memory capacity. Cutting down on external memory access in this manner also lead to more power-efficient designs. One disadvantage, however, is the increased length of the computational pipeline, which significantly affects performance for small mesh sizes.

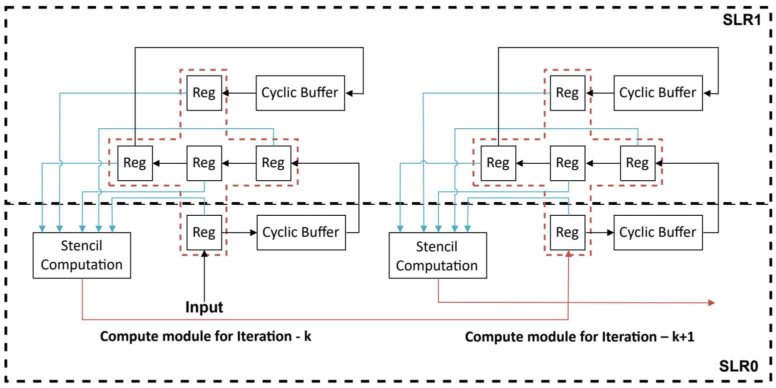


Figure 3.3: Kernel placement without SLR constraint.

3.1.3 Decoupled Kernel Pipeline

Xilinx FPGAs are partitioned into Super Logic Regions (SLRs) [90], and unrolling iterative loops multiple times can cause compute modules to span across multiple SLRs as in Figure 3.3. This can be problematic as the bandwidth between SLRs is lower than that within an SLR. To address this, we can split the compute module pipeline into several blocks, with each block becoming a separate kernel placed in single SLR as described in [39]. By doing so, prevent these kernels from spanning multiple SLRs. The kernels can then use the Advanced eXtensible Interface (AXI) stream to transfer data, as shown in Figure 3.4. This ensures that only AXI stream signals will cross the SLRs, rather than many circuit connections within compute modules.

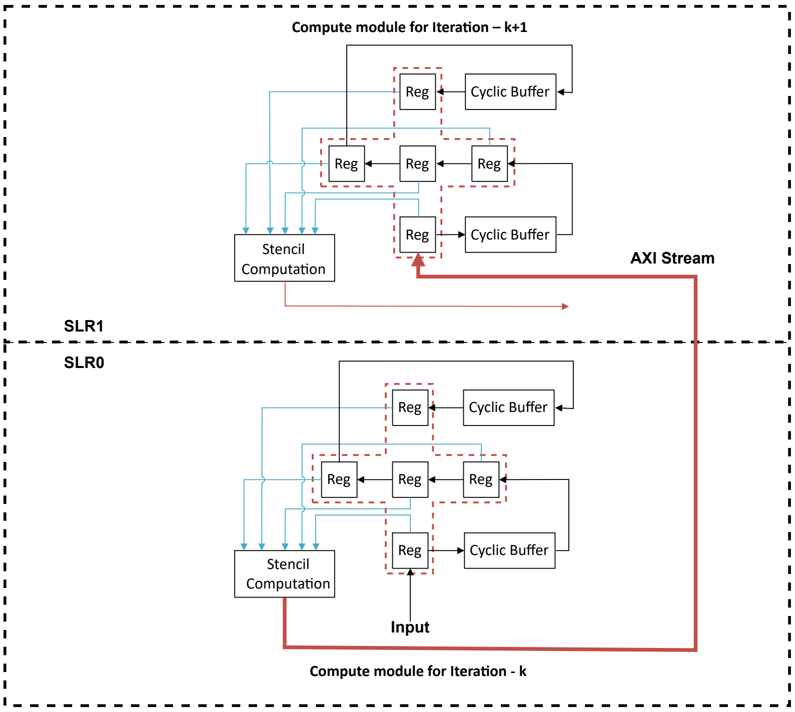


Figure 3.4: SLR constrained kernel placement.

3.1.4 Data Layout for Vector Elements

The above compute module pipeline requires input to be read from and output to be written to global memory. Real-world stencil applications require multiple meshes and mesh elements could also be vectors (see RTM application in section 3.4.3). Popular choices for the data structure of meshes with vector elements are Structure of Arrays (SoA) and Array of Structure (AoS). Choice of the data structure is closely related to data access patterns. We prefer Array of Structures (AoS) data structure if all elements in the vector block are used in stencil computation. If only some of the vector elements are used in stencil computation, then memory access can't be coalesced reducing memory throughput.

On the other hand, if the SoA data structure is assigned to single off-chip memory banks, then it would require reading chunks (large enough for better memory throughput through sequential access) cyclically of each vector index, then buffering and feeding into compute pipeline. This would make implementation a little more complex and requires additional on-chip memory for buffers. Otherwise, each index would be assigned to a separate bank in that case higher memory throughput could be obtained, but it would require more FPGA resources as multiple AXI controllers are required for each bank.

3.2 Model for Baseline Design

The performance of a baseline design, as discussed previously, therefore depends on (1) the capacity of the computational pipeline and (2) the external memory bandwidth. Computational capacity depends on the number of mesh point updates done in parallel (vectorization factor), latency of the pipeline and operating clock frequency of the FPGA. However, memory throughput depends on various factors such as the number of mesh elements transferred, and the stride between each transferred element. To simplify, we model reading/writing of contiguous data from/to memory with a maximum transfer size of 4K bytes, to reach a near optimal throughput of external/near-chip memory for the Xilinx U280 FPGA, our target hardware in this work.

Assuming that the memory throughput is sufficient to supply V mesh points (i.e. a vectorization factor of V) continuously without interruption, then the total clock cycles taken to process a row from a 2D mesh with $m \times n$ elements will be given by $\lceil \frac{m}{V} \rceil$. Here, we have padded each row to be a multiple of V if required. The compute pipeline will process $n + \frac{D}{2}$ rows as there are $D/2$ different rows between the current stencil update mesh point and farthest mesh point required for the stencil computation, where D is the stencil order. If the outer iterative loop unroll factor is given by p then the total number of clock cycles required to process the full $m \times n$ mesh for n_{iter} iterations is given by:

$$Clks_{2D} = \frac{n_{iter}}{p} \times \left(\left\lceil \frac{m}{V} \right\rceil \times (n + p \times \frac{D}{2}) \right) \tag{3.1}$$

The above extends naturally to 3D meshes as in (Equation 3.2), where the 3D mesh size is given by $m \times n \times l$ and D is then equivalent to the number of plains to be buffered.

$$Clks_{3D} = \frac{n_{iter}}{p} \times \left(\left\lceil \frac{m}{V} \right\rceil \times n \times (l + p \times \frac{D}{2}) \right) \tag{3.2}$$

As noted before, the models above only hold for cases where the vectorization factor V, which determines the number of parallel mesh points computed, does not demand more memory bandwidth than what can be supplied by the FPGA's external DDR4 bandwidth. The FPGA's HBM memory can be used to support a larger V, which could then be limited by the resources available to implement the parallel compute pipelines. An estimate of maximum V for an application can be computed by using the FPGA operating frequency f, and maximum supported bandwidth of a data channel (or port) on the FPGA, $BW_{channel}$, and the size in bytes of a mesh element $sizeof(Dtype)$ as follows:

$$BW_{channel} \geq 2Vf \times sizeof(\texttt{DType}) \tag{3.3}$$

For 2D meshes, if the width of the mesh n is a multiple of vectorization factor V, then clock cycles for computing a single mesh point (or a cell) per iteration per compute module can be obtained from equation (3.1) as :

$$Clks_{2D,cell} = 1/V + pD/2nV \tag{3.4}$$

Setting n to higher values gives a better clock cycles per mesh point ratio, the ideal being, $1/V$. But higher order stencil applications on meshes with fewer rows will have a larger $(pD)/(2nV)$ value, indicating idling in the processing pipeline. We explore techniques to reduce this idle time in Section 3.3.3.

A key parameter in (3.1) and (3.2) is the loop unroll factor, p which directly determines performance, where a large p reduces the total clock cycles required. However, p is limited by the available resources on the FPGA as in Fig. 3.2, a larger p requires more DSP blocks and LUTs. Furthermore, the internal memory required for a compute module, primarily due to memory capacity for the cyclic buffers also determines p. The number of DSP blocks required for a single mesh-point update, G_{dsp} depends on the stencil loop kernel's arithmetic operations and number representation. Here we consider single precision floating point arithmetic. With a V vectorization factor, the total consumed is $V \times G_{dsp}$. If the total available DSP blocks on the FPGA is $FPGA_{dsp}$ then the maximum unroll factor based on DSP resources, p_{dsp} is given by:

$$p_{dsp} = FPGA_{dsp}/VG_{dsp} \tag{3.5}$$

The internal memory requirement for a single compute module which performs a D order stencil operation on an $m \times n$ mesh is $D \times m$. If the total available internal memory on the device is $FPGA_{mem}$, then maximum possible iterative unroll factor based on internal memory requirements, p_{mem} is :

$$p_{mem} = \frac{FPGA_{mem}}{sizeof(\texttt{DType}) \times Dm} \tag{3.6}$$

Here, k is the size of a mesh element in bytes. The denominator of (3.6) becomes $kDmn$ for 3D meshes. Thus we see that the internal memory of an FPGA, directly limits the solvable mesh size. Usually, the above ideal depth is not achievable, as the FPGA internal memory, BRAMs and URAMs, are quantized (for example BRAMs are 18Kb/36Kb and URAMs are 288Kb on the U280). Additionally, the limited width configurations of the URAMs, plus the need to allow for flexible routing further reduces the effective internal memory resources. Thus we usually target an 80%–90% internal memory utilization. Then the maximum iterative loop unroll factor is given by the minimum of p_{dsp} and p_{mem}. It is also worth considering that a larger pipeline depth, and hence more resource consumption leads to the design spreading over multiple SLRs. Communication between SLRs increasing routing congestion between these regions, directly impacting the achievable operating frequency.

3.3 Optimizations

Further optimizations and extensions are required to obtain high throughput for more complex applications. These include (1) spatial and temporal blocking, specifically for solvers over larger meshes, and (2) batching for improving performance and throughput of stencil applications on smaller meshes. In this section, we build on the baseline design from Section 3.1 and extend the performance models to account for these optimizations.

3.3.1 Spatial and Temporal Blocking

The baseline design attempts to obtain perfect data reuse, requiring FPGA internal memory (consisting of BRAMs and URAMs) to be of size $D \times m$ for 2D and $D \times m \times n$ for 3D meshes. Equation (3.6) illustrates this, where the requirement becomes highly limiting for applications with higher order (D) stencils and/or on larger meshes (increasing m). Even if the mesh fully fits in the FPGA's DDR4 memory, a sufficiently large mesh could result in a p_{mem} less than one, meaning that even a single compute module cannot be synthesized. A solution is to implement a form of spatial blocking, similar to cache blocking tiling on CPUs, for the FPGA.

The idea is to use the baseline design to build an accelerator that operates on a smaller block of mesh elements and then transfer one such block at a time to the compute pipeline from FPGA DDR4 memory. The compute pipeline is designed with an appropriate vectorization factor (V) and an outer iterative loop unroll factor (p). Larger p results in better exploitation of temporal locality, where the execution uses the same data several times. One issue with such a blocked execution is when applying the computation over the boundary of a block where a stencil computation on the boundary will not have the contributions from all the neighboring elements in the mesh as in Figure 3.5. The solution is to overlap blocks such that the correct computation is carried out on the boundary by a subsequent block. The amount of overlap depends on the order of the stencil. Overlapping leads to redundant computation. However this overhead can be acceptable, due to the savings from further exploiting local data in multiple iterations where overlapped region will widen.

The main challenge of tiling then is to get close to maximum DDR4 memory bandwidth, due to the latency of smaller, non-contiguous data transfer sizes. Such data transfers results due to a strided access pattern in one dimension when accessing memory locations within a spatial block. For example on the Xilinx U280, it takes 16 clock cycles to transfer 1024 Bytes via the 512 bit wide AXI interface bus, but the latency of the

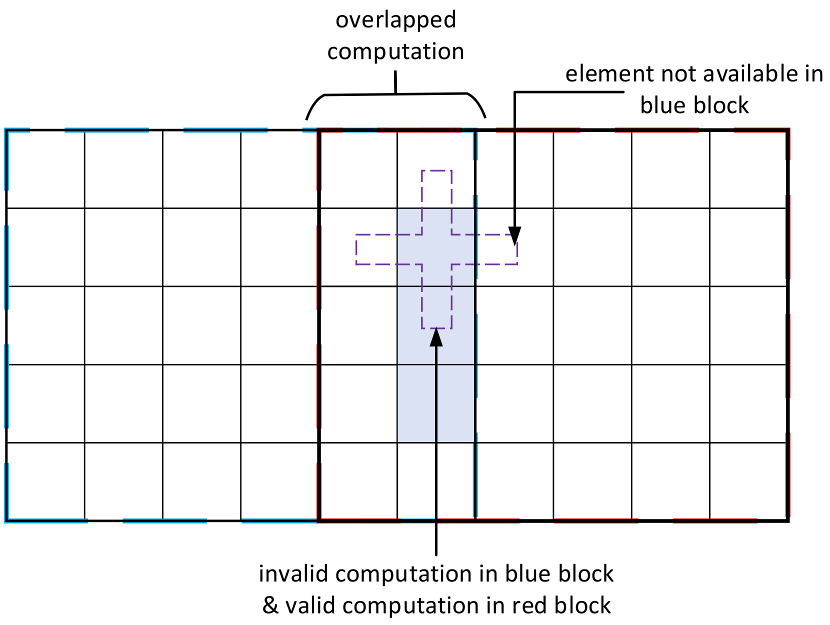

overlapped computation
element not available in blue block
invalid computation in blue block & valid computation in red block

Figure 3.5: Overlapped spatial blocks.

transfer is about 14 clock cycles. As such, multiple read/write requests should be made to hide the latency of each individual memory transaction. The preference to maintain a 512 bit wide bus interface to obtain better memory bandwidth further increases the amount of redundant computation at block boundaries as we must maintain a 512 bit alignment in read/write transactions, regardless of the order of the stencil.

A final modification is the need to loop through the spatial blocks to solve over the full mesh. This control loop is best implemented on the FPGA to reduce latency due to the host calling multiple kernels on the FPGA. An important consideration is finding the optimal block size and its offset from the start of the mesh. The block size and offsets need only be computed once, which can be done on the host and copied to FPGA memory. Considering a 3D stencil application over a mesh of size $m \times n \times l$ solved by computing over with blocks (or tiles) of size $M \times N \times l$, the valid number of mesh points computed per block is given by:

$$Block_{valid} = (M - pD) \times (N - pD) \times l \tag{3.7}$$

Since the number of clock cycles required to process p iterations (or a temporal block) on the $M \times N \times l$ spatial block is similar to the baseline design, the average time taken to compute one block (assuming block dimensions are a multiple of V) would be:

$$Clks_{block,3D} = \frac{M}{V} \times N \times \frac{l + pD/2}{p} \tag{3.8}$$

Dividing (Equation 3.7) by (Equation 3.8) leads to the number of valid mesh points (or cells) computed per clock cycle (i.e. throughput, T) :

$$T = (1 - \frac{pD}{M}) \times (1 - \frac{pD}{N}) \times (\frac{pVl}{l + pD/2}) \tag{3.9}$$

Now, substituting N from Equation 3.6, for a 3D application, assuming full utilization of the FPGA's internal memory by a block, it can be shown that maximum throughput can be achieved for a given p when:

$$M = \sqrt{\frac{FPGA_{mem}}{sizeof(\texttt{DType}) \times pD}} \tag{3.10}$$

The corresponding N, can be shown to be also equal to M, implying a square block to give the best throughput. However, the throughput also varies with p and this can be analyzed by considering a square tile (i.e. $M = N$) applied to equation (3.9) and assuming l to be very large such that $\frac{l}{l+pD/2}$ is close to 1. With these assumptions, we can show that maximum throughput is achieved, for a given M, when setting p to a p_{max} given by:

$$p_{max} = M/3D \tag{3.11}$$

Obtaining a value for pV from Equation 3.5, assuming we use all the computational capacity of the FPGA, we can rewrite Equation 3.9 as:

$$T_{3D} = (1 - \frac{pD}{M})^2 \times \frac{FPGA_{dsp}}{G_{dsp}} \times (\frac{l}{l + pD/2}) \tag{3.12}$$

The same for a 2D stencil application can also be derived as:

$$T_{2D} = (1 - \frac{pD}{M}) \times \frac{FPGA_{dsp}}{G_{dsp}} \times (\frac{n}{n + pD/2}) \tag{3.13}$$

Here, we see that reducing pipeline depth p and increasing V will improve the performance of the spatial blocked design. The effect of p is more significant for 3D applications.

3.3.2 Spatially Blocked Design using Multiple HBM Ports

Redundant computation in a spatially blocked design is proportional to the iterative loop unroll factor p. Considering that, a larger possible value for V is preferred to make iterative loop unroll factor p a lower value while getting a similar amount of mesh updates per clock (Equation 3.5). Modern FPGAs comes with High Bandwidth Memorys (HBMs) and multiple HBM banks should be utilized to scale the V. Choice of the data layout with respect to HBM banks also determines the obtainable memory throughput. The usual choices are cyclic partitioning and block partitioning. Block partitioning maps each block in the data structure to an HBM bank. cyclic assignment assigns one or more mesh elements cyclically to HBM banks.

As vectorized stencil computation requires multiple adjustant mesh points fed to com-

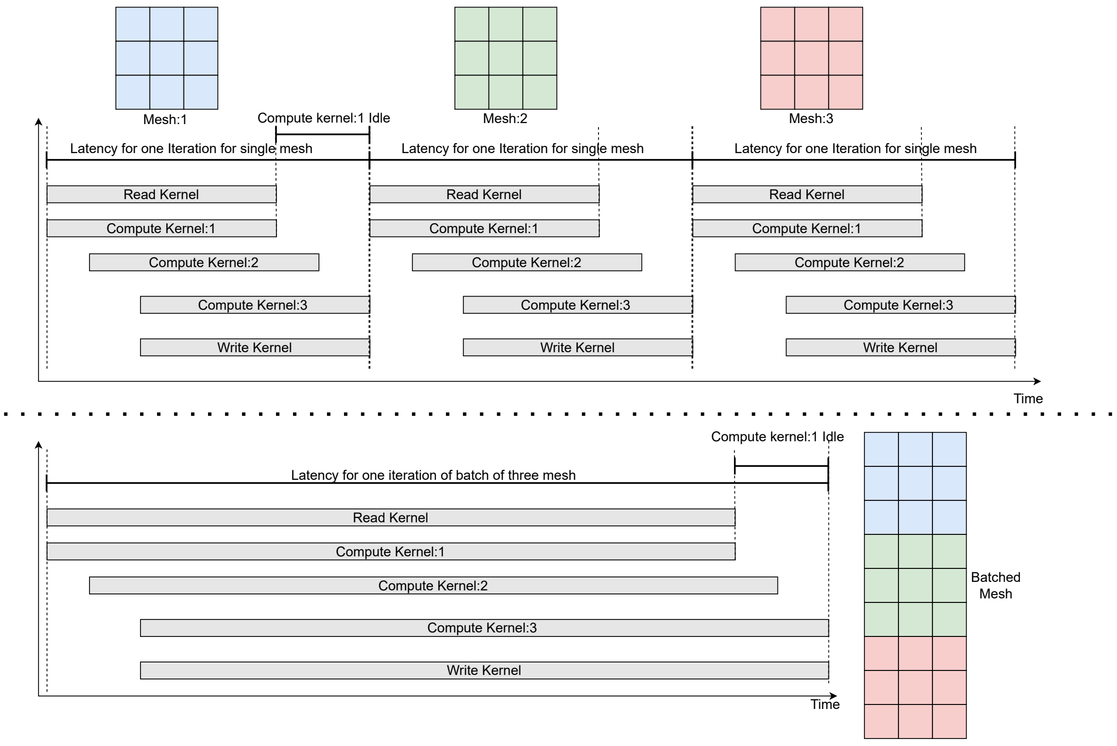


Figure 3.6: Individual Vs Batched computation.

pute pipeline at the same clock. Block partitioning will limit this as all adjustant mesh points will be in the same bank. We prefer smaller blocks (Lets say 4/8/16) of mesh elements cyclically assigned to each bank as wider port memory access can bring multiple mesh elements at a clock cycle. We also prefer padding the first dimension of the mesh with zeros such that it completes cyclic partitioning with targetted number of HBM banks. This would help to avoid complex multiplexer to make coalesced memory access and complex addresses calculation when reading each tile block in spatially blocked design.

3.3.3 Batching

A final optimization attempts to improve throughput for smaller mesh problems that usually perform poorly on accelerator platforms, including FPGAs. On traditional architectures such as GPUs the reason is the the under-utilization of the massive parallelism available. Essentially the time spent calling a kernel on the device and the overheads for data movement between host and device comes to dominate the actual processing time.

On an FPGA, in addition to the above, further overheads are caused due to the latency of the processing pipeline, as given in equation (3.4), compared to the time to process the mesh. The active time of compute modules and their idle time is illustrated in Figure 3.6. The idle time is proportional to the width of the 2D mesh. Thus if a large number of smaller meshes are to be solved, as is the case in financial applications [64], then processing one mesh at a time incurs significant latencies. This motivates the idea of grouping together meshes with the same dimensions in batches, increasing the overall throughput of the solve as illustrated in Figure 3.6. Here, meshes are extended in the last dimension by stacking up the small meshes. Now, the inter-compute module latencies only occur once at the start of the batched solve. With B, 2D meshes in a batch, the time to process a single mesh within a batched execution is given by:

$$Clks_{2D/batched\_mesh} = \left(\left\lceil \frac{m}{V} \right\rceil \times (n + p \times \frac{D}{2B}) \right) \tag{3.14}$$

Thus, increasing B significantly reduces the idle time from (3.4). Similar reasoning can be applied for batched 3D meshes. Here we note that, last dimension batching doesn't require additional key resources, on-chip memory and DSP Units. Since a new outer loop is introduced / bound of flattened loop is increased, minor increase in clock critical path can be observed. The `ceil` operation in Equation 3.14 becomes significant overhead for meshes with smaller width on implementation with larger V. This overhead can be eliminated by first Dimension batching, where elements of V meshes comes one after another. Since window buffers keep the elements of V meshes, on-chip memory requirement will be V times of baseline design. Hybrid batching, combining first dimension and last dimension batching can be used to get better throughput on FPGAs.

This workflow assumes that part of the program accelerated on FPGA can be mapped to data-flow graph based computation. Lets assume the part of the program to be accelerated contains the multiple stencil computational loops operating on multiple data structures. on FPGA, these stencil loops will be mapped to computing nodes. data structures will be introduced to data flow graph by read modules and will be taken out from data flow graph using write modules. Delay buffer will be another nodes in data-flow graph to avoid stalling in data flow due to delay in processing in some computing nodes. Here we target parallel execution of all compute nodes on FPGAs, which can be effectively utilised by batching or meshes with larger sizes. Proportional data flow is targeted to avoid stalling and idling of a compute node, Hence proportional vectorization factor V and loop initiation interval `II` is required. In Xilinx FPGAs this data-flow graph should be equally partitioned and to be mapped to SLRs.

Fitting such a data-flow graph on FPGA comes with constraints, FPGA's resources are limited as well as available bandwidth. Design space exploration can be done on following parameters, iterative loop unroll factor p, Vectorization factor V and loop initiation interval `II`. $dims$ maximum mesh dimension targeted for the application. Lets assume $R_{dsp}(p, V, II)$ is the DSP requirement function, $R_{BW}(p, V, II)$ is bandwidth requirement function and $R_{onchip}(p, dims)$ is on chip memory requirement function.

Table 3.1: Experimental system's specifications.

FPGA	Xilinx Alveo U280 [91]
DSP blocks	8490
BRAM / URAM	6.6MB (1487 blocks) / 34.5MB (960 blocks)
HBM	8GB, 460GB/s, 32 channels
DDR4	32GB, 38.4GB/s, in 2 banks (1 channel/bank)
Host	Intel Xeon Silver 4116 @2.10GHz (48 cores)
	256GB RAM, Ubuntu 18.04.3 LTS
Design SW	Vivado HLS, Vitis-2019.2
GPU	Nvidia Tesla V100 PCIe [91]
Global Mem.	16GB HBM2, 900GB/s
Host	Intel Xeon Gold 6252 @2.10GHz (48 cores)
	256GB RAM, Ubuntu 18.04.3 LTS
Compilers, OS	nvcc CUDA 9.1.85, Debian 9.11

3.4 Performance

In this section we apply the FPGA design strategy, optimizations, and extensions to illustrate their utility in accelerating stencil computations for explicit-iterative numerical solvers. We select three representative applications consisting of, both 2D and 3D, low and high order, and with single and multiple stencil loops to explore the versatility of our design flow. Model-predicted resource utilization estimates are used to determine initial design parameters, and runtime performance is compared to model predictions for each application. The implementations target the Xilinx Alveo U280 accelerator board and demonstrate concrete implementations for each application. We use Vivado C++ due to ease of use for configurations, arbitrary precision data types, and support of some C++ constructs compared to OpenCL, but note OpenCL can be equally used to implement the same design. Additionally, we compare equivalent implementations of each application's performance on a modern GPU system for comparison[1] (raw runtime values are available in Appendix section C.1) . Table 3.1 briefly details the specifications of the FPGA and GPU systems (both hardware and software) used in our experiments.

Table 3.2: Model parameters for baseline and batched designs.

Application	Freq.	G_{dsp}	p_{dsp}	
	(MHz)		(model)	(actual)
Poisson-5pt-2D	250	14	68	60
Jacobi-7pt-3D	246	33	28	29
Reverse Time Migration	261	2444	3	3

[1]We have omitted CPU performance results here as our previous work [64] shows that GPUs provide significant speedups over CPUs for these applications

Table 3.3: Spatial blocking model parameters.

App.	p	V	M	N	$T_{2D\|3D}$	Valid ratio
Poisson-5pt-2D	60	8	8192		472	98.5%
Jacobi-7pt-3D	3	64	768	768	189	98.4%

3.4.1 Poisson-5pt-2D

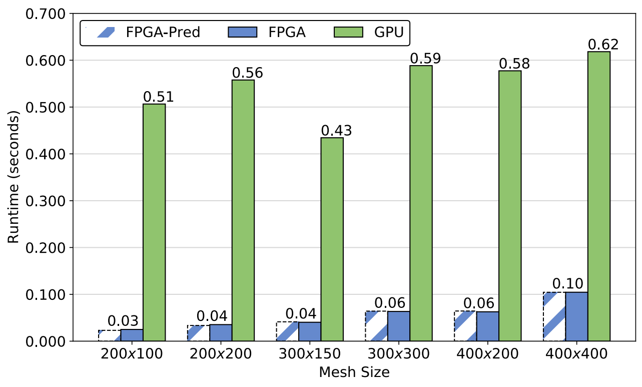


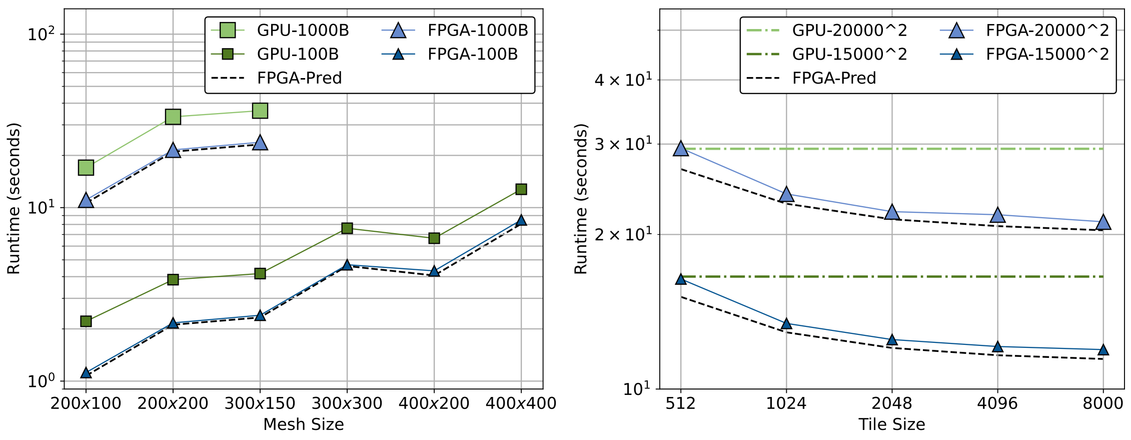
Figure 3.7: Poisson-5pt-2D performance (Baseline - 60k iters).

The first application is a 2D Poisson solver which uses a 2nd order stencil, with scalar elements:

$$U_{i,j}^{t+1} = \frac{1}{8}\left(U_{i-1,j}^{t} + U_{i+1,j}^{t} + U_{i,j-1}^{t} + U_{i,j+1}^{t}\right) + \frac{1}{2}U_{i,j}^{t} \tag{3.15}$$

A suitable initial vectorization factor V can be identified by using (3.3) and assuming an operating frequency of 300MHz given this is the default set by the Vivado HLS tools. For a baseline implementation of Poisson a value of 8 for V is calculated when using a single DDR4 channel or two HBM channels with a frequency of 300MHz. However, this frequency could only be supported when iterative loop unroll factor p is in the order of 1–20. Higher p lead to routing congestion, which limited achievable frequency. As such the frequency was reduced to 250MHz to support a p of 60, which we observed to give the best performance for this stencil. We find in some cases such a trial frequency adjustment is unavoidable, but our model significantly narrows the design space, enabling us to reason about and quickly obtain an optimal configuration. The number of DSP blocks required for a single mesh-point's stencil computation for Poisson and the resulting p_{dsp} from (3.5) for $V = 8$, assuming a 90% DSP utilization, is given in the first row of Table 3.2.Column 4 gives the predicted p_{dsp} from our performance model, while column 5 is the actual result after synthesis, indicating good agreement with the predicted design.

Figure 3.7 and Figure 3.8 (a) present the runtime performance of Poisson-5pt-2D, with the above design and compare the resultant performance to an equivalent implementation on the Nvidia V100 GPU. The achieved bandwidth and energy consumption from these runs are summarized in Table 3.4. The bandwidth is computed by counting the total



(a) Batching - 60k iters

(b) Spatial-blocking - 6k iters

Figure 3.8: Poisson-5pt-2D performance.

number of bytes transferred during the execution of the stencil loop (looking at the mesh data accessed) and dividing it by the total time taken by the loop. Baseline FPGA performance is significantly better than on the V100, since the GPU is not saturated by this application. The batching of 2D meshes as in [64] improves GPU performance significantly and offers a closer comparison. The FPGA achieves a maximum speedup of about 30–34% for different mesh sizes and batching sizes of 100 (100B) and 1000 (1000B). Memory bandwidth results indicate high utilization of the communication channels in agreement with the observed runtimes. The `xbutil` utility was used to measure power during FPGA execution, while `nvidi-smi` was used for the same on the V100. The power consumption of the FPGA during the 1000B runs is indicative of the significant energy efficiency of the device compared to a GPU. The FPGA was operating at an average 70W, while the GPU's power consumption ranged from 40W (for single batch) to 210W for 1000B runs on the larger meshes.

Table 3.4: Poisson-5pt (Baseline and Batched, 60k iters) - Bandwidth (GB/s) and Energy(kJ).

Mesh	Baseline		100B		1000B		Energy-1000B	
	FPGA	GPU	FPGA	GPU	FPGA	GPU	FPGA	GPU
200×100	384	18	857	404	867	530	0.77	3.48
200×200	543	32	886	465	892	540	1.50	6.74
300×150	535	38	901	483	907	560	1.66	7.60
300×300	681	69	922	530				
400×200	612	62	889	536				
400×400	735	116	904	560				

To implement Poisson-5pt-2D on larger meshes with spatial blocking, we assume a V and p equivalent to the baseline design and compute the valid mesh points updated per clock cycle using (3.12). Here we assume the dimensions of the mesh to be very large.

Table 3.5: Poisson-5pt (Spatial-blocking, 60k iters) - Bandwidth (GB/s) and Energy(kJ).

Mesh	Tile Size	BW		Energy	
		FPGA	GPU	FPGA	GPU
15000^2	1024	805	607	0.93	2.91
	4096	892		0.84	
	8000	905		0.83	
20000^2	1024	800	609	1.67	4.96
	4096	879		1.52	
	8000	907		1.48	

Table 3.3 lists the model parameters for spatial blocking. For Poisson we see that the 2D spatially blocked designs theoretically perform similar to the baseline design and thus we need not change the compute pipeline. Runtime, bandwidth and energy consumption of this implementation is given in Figure 3.8 (b) and Table 3.5, respectively, including comparison to performance from the V100 GPU. Again we see good speedups and higher energy efficiency achieved with the FPGA, this time on large problem sizes with tiling.

3.4.2 Jacobi-7pt-3D

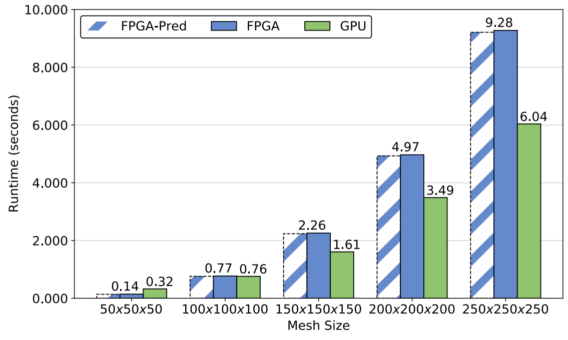
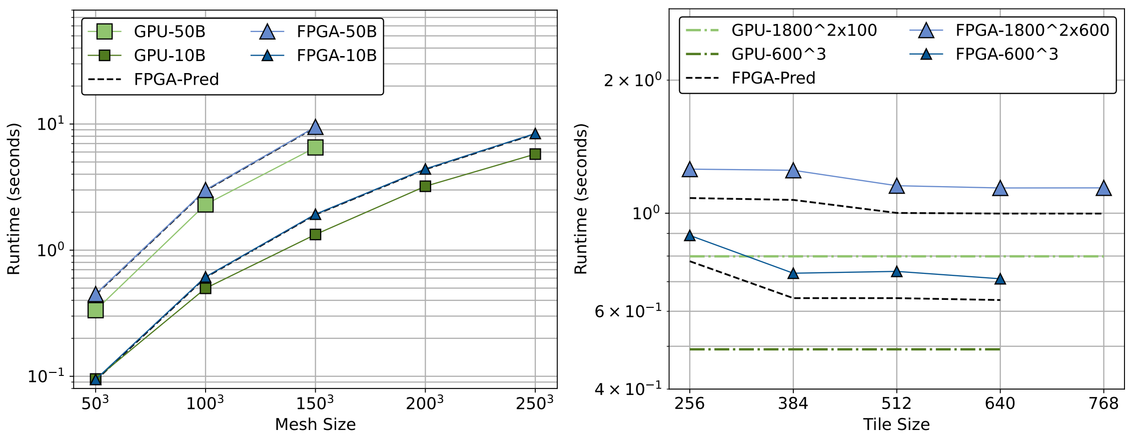


Figure 3.9: Jacobi-7pt-3D performance (Baseline - 29k iters).

The Jacobi iteration as a 3D, 7-point stencil, provides us with an initial, 3D, single stencil loop, for our evaluation:

$$\begin{aligned} U^{t+1}_{i,j,k} = k_1 U^t_{i+1,j,k} + k_2 U^t_{i-1,j,k} + k_3 U^t_{i,j-1,k} + k_4 U^t_{i,j,k} + \\ k_5 U^t_{i,j+1,k} + k_6 U^t_{i,j,k+1} + k_7 U^t_{i,j,k-1} \qquad (18) \end{aligned}$$

This application requires higher internal memory for the baseline design. For the spatially blocked design it involves transfers less than 4K from memory, which makes it difficult to approach raw external memory bandwidth. This is different to the baseline/-batched and 2D spatially blocked design. We speculate that this could be the reason for the slightly less accurate model predictions in Figure 3.10(c). While the stencil is still fairly simple, now we see the GPU outperforming the FPGA conclusively, in both baseline, Figure 3.9 and batched Figure 3.10(a) tests. The V100 GPU gives nearly 40% faster



(a) Batching - 2900 iters (b) Spatial-blocking - 120 iters

Figure 3.10: Jacobi-7pt-3D performance.

Table 3.6: Jacobi-7pt-3D (Baseline and Batched): Bandwidth (GB/s) and Energy(kJ)

Baseline (29k iters) and Batching (2.9k iters)								
Mesh	Baseline		10B		50B		Energy-50B	
	FPGA	GPU	FPGA	GPU	FPGA	GPU	FPGA	GPU
50^3	202	83	307	284	323	404	0.04	0.07
100^3	301	284	378	434	387	469	0.27	0.51
200^3	374	496	421	548	426	543	1.96	3.77
250^3	391	559	431	585				
300^3	403	553	438	569				

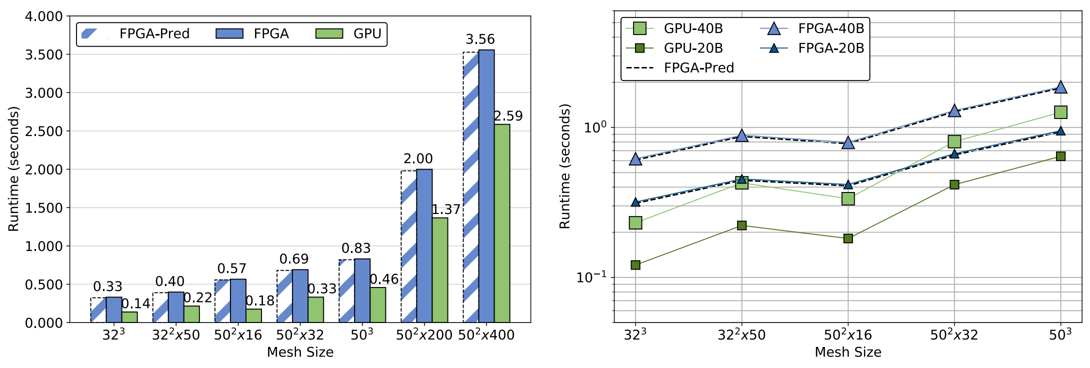
runtimes on the 50B problem. However, the FPGA remains more energy efficient for the same problem. For the 200×200 problem with 50B, it is nearly $2\times$ more energy efficient than the faster GPU run (see Table 3.6). The FPGA operated at an average 90W while the GPU power ranged from 77–240W. Spatial blocking was significantly more challenging and the resulting FPGA design, using a 640^2 tile size was about 40% slower than the GPU runtime (see Figure 3.10(b)). However, the FPGA was again more energy efficient, operating at an average 70W consuming about 40–50% less energy than the GPU (operating at 180–216 W) as seen in Table 3.7.

3.4.3 Reverse Time Migration (RTM) - Forward Pass

The final application we applied our development flow to is the forward pass from a Reverse Time Migration (RTM) solver [11]. The application represents algorithms of interest from industry [16], going beyond simple single stencil loops. It includes an iterative loop consisting of multiple stencil loops as summarized in Algorithm 3. Y, T and $K_1..K_4$ are 3D floating-point (SP) data arrays defined on the mesh consisting of vector elements of size 6. Y holds current values and T holds intermediate values, both updated with the f_{pml} function which uses a 25-point, eighth order 3D stencil. $K_1..K_4$ is accessed/updated

Table 3.7: Jacobi-7pt-3D (Spatial-blocking, 120 iters): Bandwidth (GB/s) and Energy(kJ)

Mesh	Tile Size	BW		Energy	
		FPGA	GPU	FPGA	GPU
600^3	256	233	392	0.062	0.106
	512	281		0.051	
	640	292		0.049	
$1800 \times 1800 \times 100$	256	247	363	0.088	0.143
	512	270		0.080	
	640	273		0.079	



(a) Baseline - 1800 iterations

(b) Batching - 180 iterations

Figure 3.11: RTM performance.

with a self-stencil (or zeroth-order, i.e. i, j, k). ρ and μ are two 3D scalar coefficient meshes, which are also accessed using a self-stencil. Array of structures (AoS) data layout is used in FPGA implementation as data access is sequential on FPGA and AoS requires less logic for read module implementation on FPGA. On GPU, we tried both AoS and SoA (Structure of Arrays) and better performance is observed when using SoA data structure. We speculate better access data pattern when using SoA as the reason.

Algorithm 3: RTM - Forward Pass

```
1: for $i = 0,\ i < n_{iter},\ i++$ do
2:     $K = f_{pml}(Y_{25pt}, \rho, \mu) \times dt;\ T = Y + K/2;\ \ S = K/6$
3:     $K = f_{pml}(T_{25pt}, \rho, \mu) \times dt;\ T = Y + K/2;\ \ S = S + K/3$
4:     $K = f_{pml}(T_{25pt}, \rho, \mu) \times dt;\ T = Y + K;\ \ S = S + K/3$
5:     $K = f_{pml}(T_{25pt}, \rho, \mu) \times dt;\ Y = Y + S + K/6$
6: end for
```

This application is significantly more complex than the previous applications and pushes the resource usage on the FPGA to its limits. Nevertheless our design strategy is able to provide a good implementation, albeit limited to a batched design. The number of stencil loops was reduced by fusing the K_1, K_2, and K_3 with the corresponding T loop. The K_4 and final Y update were merged into one further loop, resulting in a total of 4 loops. For an FPGA implementation, all the four fused loops needed to be brought into a single pipeline. Intermediate data T and $K_1 ... K_4$ were replaced with a FIFO stream connected through window buffers. Similarly ρ, μ and Y were internally buffered and fed to subsequent compute units. These optimizations reduce the number of memory accesses to a single read and write of Y and a single read each for ρ and μ. These are significant savings compared to the original loop chain.

A limitation of the FPGA implementation is that the mesh plane size (in this 3D application), is limited to 64^2 as it uses 3D stencils on a 6 dimensional element (i.e. a vector of 6 floats). Furthermore, partitioning four compute-intensive kernels on the U280's three SLR regions was a significant challenge. Our implementation avoids spanning of a compute unit on multiple SLRs to avoid inter SLR routing congestion, by setting V to 1, allowing us to fit the four fused loops in one SLR. This, then allows for an iterative loop unroll factor of 3 (p) given the three SLRs on the U280. We do note that using more HBM channels could provide more bandwidth to obtain a larger V, but we have not explored this in current work. A solution for the limited mesh size is of course spatial blocking, but it requires $p = 4$. This leads to a tile size dimension $M = 96$ from (3.11) given D is 8, which requires a large amount of FPGA internal memory, making an implementation on the U280 challenging as the four fused loops will span across SRLs. We leave this to future work.

Table 3.8: RTM - Average Bandwidth (GB/s) and Energy(kJ)

Baseline (1800 iters) and Batching (180 iters)

Mesh	Baseline		20B		40B		Energy-40B	
	FPGA	GPU	FPGA	GPU	FPGA	GPU	FPGA	GPU
$32 \times 32 \times 32$	95	227	197	518	203	542	0.043	0.037
$32 \times 32 \times 50$	120	222	211	430	215	447	0.062	0.062
$50 \times 50 \times 16$	68	221	186	426	195	463	0.055	0.056
$50 \times 50 \times 32$	105	219	218	350	224	362	0.091	0.137
$50 \times 50 \times 50$	134	243	233	346	238	352	0.130	0.215

From the runtime results in Figure 3.11 and bandwidth results in Table 3.8 we see that the FPGA implementation is giving a competitive performance to the GPU. Note that, given there are four stencil loops pipelined on the FPGA, the bandwidth reported is for the pipelined loop chain. The GPU bandwidth, therefore, is the average for the full loop chain. GPUs bandwidth reaching upto 518 GB/s indicate the optimal implementation and a slight drop in bandwidth for larger mesh sizes can be seen. Higher order stencils on larger meshes will require larger cache for full data reuse, we speculate required cache lines getting evicted is reason for lower average bandwidth for larger meshes. Again we see that the FPGA operates at a lower average power (70W) than the GPU (51–170W) and FPGA saves 40% of energy on GPU based computation for largest configuration.

3.5 Concluding Remarks and Discussion

In this Chapter we developed a unified workflow and a supporting predictive analytic model for FPGA synthesis of structured-mesh stencil applications that combines standard state-of-the-art techniques with a number of high-gain optimizations targeting features of real-world work loads. The model allows estimation of design parameters, resource usage, and performance for performant FPGA implementation. The workflow was applied to three representative applications, implemented on a Xilinx Alveo U280 FPGA. Performance was compared to highly-optimized HPC-grade Nvidia V100 GPU code. In most cases, the FPGA is able to match or improve on GPU performance. However, even when runtime is inferior to the GPU, significant energy savings, over 40% for the largest application, are observed. Estimations produced by the model were shown to be accurate and a good guide in the design process. Future work will investigate how a similar workflow can be applied to implicit solvers and further automating the development of this class of application on FPGAs, including alternative numerical representations. The FPGA and GPU source code developed in this Chapter are available at [37].

Chapter 4

Implicit Schemes on FPGAs

In the preceding Chapter, we introduced a design space exploration and optimal workflow for explicit applications based on structured meshes on Xilinx FPGAs. However, since structured mesh-based implicit numerical schemes offer faster convergence and better numerical stability benefits, this Chapter aims to explore their acceleration on FPGAs, particularly schemes that utilize tridiagonal system solvers. Previous research works have implemented popular tridiagonal system solver algorithms using both HDL [54, 87, 93] and high-level languages [88, 49, 47, 48] on FPGAs. However, their focus has mainly been on solving single tridiagonal system solvers, rather than analyzing the characteristics of the application class with respect to the architectural capabilities of FPGAs for implicit solvers commonly found in real-world applications.

In this Chapter, we evaluate different tridiagonal solver algorithms on FPGAs for batched systems using the analytical models. Based on such analysis, we designed a batched thomas solver library to solve small and medium tridiagonal systems and a Novel Tiled Thomas solver library for solving larger tridiagonal systems. These two tridiagonal system solvers are implemented as a template-based HLS library for implicit application implementation on FPGAs. New library demonstrates over one magnitude performance improvement compared to Xilinx library, for larger batches of tridiagonal systems.

Two non-trivial implicit applications were developed using this new library and explicit stencil solver techniques presented in the previous Chapter. Several dataflow optimization techniques and memory access transformation techniques for this class of applications are presented using the two representative applications. We further scale the performance by taking advantage of HBM memory in modern FPGAs. Performance predicted by analytical models developed in this Chapter reaches over 85% accuracy. A detailed comparison using a current state-of-the-art GPU library for multi-dimensional tridiagonal systems for these two applications on an Nvidia V100 GPU shows the FPGA achieving competitive or better runtime and significant energy savings of over 30%. Through these applications, This Chapter lessons about the types of applications where FPGAs can challenge the current dominance of GPUs.

4.1 FPGA Design

4.1.1 Small and Medium System Solvers

Considering the resources available on an FPGA, a single tridiagonal system solve, using the Thomas algorithm in Algo. 1 would require 4 multiplications, 1 division, and 2 subtractions for the forward path and one multiplication and subtraction for the backward path. However, due to dependencies for computing d_i^* and c_i^*, each iteration of the forward path loop must be executed serially, incurring the full arithmetic pipeline latency, l_f ($\approx$30 clock cycles on a Xilinx U280 FPGA for FP32), to complete the forward loop datapath. Additionally the backward loop can only start when all iterations of the forward path have been completed, due to the reverse data access where the loop starts from iteration $N-2$. Thus the total latency for solving a single system with the Thomas algorithm would be approximately $l_f \times N + l_b \times N$ clock cycles (assuming l_b cycles is the arithmetic pipeline latency for completing a single iteration of the backward loop). On the other hand, a PCR based single solver implementation would require 4 subtractions, 9 multiplications, and 1 division within the inner loop of Algorithm 2. If l is the arithmetic pipeline latency of the inner loop, then the total number of clock cycles for the PCR algorithm, is $(N+l) \times logN$. Here we assume that the outer loop is executed serially and a fully pipelined inner loop, i.e., an initiation interval of one. Given inner loop iterations are independent, they can be unrolled by some factor $f_U = 2, 3, ...$ which will then require $f_U\times$ the resources to implement the inner loop. The total clock cycles consumed will then be $(N/f_U + l) \times logN$. The outer loop iterations have a dependency and thus cannot be unrolled.

For the Thomas solver, there are l_f clock cycles between consecutive iterations of a single system solve in the forward path. This can be considered as a *dependency distance*. As such, we could attempt to solve l_f tridiagonal systems to fully utilize the forward path circuit pipeline. This can be done by *interleaving* the iterations of the forward pass loop of the Thomas solver such that iteration 1 of system 1 is input followed by iteration 1 of system 2 and so on, per clock cycle, up to iteration 1 of system l_f. In fact selecting a group, $g = MAX(l_f, l_b)$ enables g system solves to be interleaved, saturating the pipeline. If there are B total tridiagonal systems to be solved, i.e. a batch size of B, then the total latency with Thomas is given by (4.1):

$$(3 + \lceil B/g \rceil) \times gN \tag{4.1}$$

Thus for large B the total latency tends to BN. This is a characteristic of all $\mathcal{O}(N)$ algorithms, which can ideally be pipelined to accept inputs each clock cycle at the cost of increased resource consumption.

For the PCR algorithm, there are no dependencies between iterations of a single system and solving a batch of B systems (by batching the inner loop) incurs the latency in (4.2), here l_{il} is the pipeline latency of the inner loop:

$$(BN/f_U + l_{il}) \times logN \tag{4.2}$$

For large B, dividing (4.2) by (4.1) gives a factor of $logN/f_U$ pointing to the fact that the batched Thomas solver is $logN$ times faster than batched PCR, for $f_U = 1$. Thus, to match the Thomas solver latency, a batched PCR implementation needs an unroll factor $f_U = logN$. However, given that the PCR inner loop has a considerably higher resource requirement compared to the Thomas solver, the batched Thomas solver will always provide better performance for the same amount of FPGA resources. An exception to this is when the system size, N, is large and FPGA on-chip memory becomes the limiting factor. Designs for such cases are discussed in Sec 4.1.2.

Considering a batched solver based on the SPIKE algorithm, assume each system in the batch is of size N. The algorithm creates N_b blocks and each has LU and UL factorization done in parallel, followed by the pentadiagonal solve and then back-substitution in parallel. This incurs a total latency given by (4.3):

$$(3 + \lceil BN_b/g \rceil) \times gN/N_b + N_bC + 3 \times gN/N_b \tag{4.3}$$

The latency for the factorization for each block (first term), is similar to a Thomas forward and backward solve carried out in an interleaved manner. Although the number of clock cycles spent on the pentadiagonal reduced system solve is BN_bC (assuming a linear latency model), only the latency for first stage of the pentadiagonal solver is added to equation 4.3 as all three modules are pipelined. The final term is the added delay due to back-substitution stage which is again a Thomas solver. When B is sufficiently large and stages are pipelined, a latency of BN is achieved. Again this is due to the SPIKE algorithm having a $\mathcal{O}(N)$ complexity. However, if $BN_bC \geqslant BN$ then dataflow must stall for some time, decreasing throughput. Resource consumption of the LU/UL factorizations requires $3\times$ the resources for an equivalent Thomas solver and the pentadiagonal solver needs additional resources, again more than an equivalent Thomas solver.

Given the lower resource requirements and profitability of the Thomas algorithm, compared to the other algorithms, we first focus on its optimized batched implementation on an FPGA for system sizes that can fit into on-chip memory. As we are interleaving groups of g, the c_{i-1}, d_{i-1} and u_{i+1} values needs to be stored in on-chip memory such that they can be used in subsequent (i^{th}) iterations. For a FP32 implementation we have found that a grouping of 32 is sufficient to effectively pipeline the computation (this is 64 for FP64) on the Xilinx Alveo U280. The forward and backward loops operate in opposite directions and thus a First-In-First-Out (FIFO) buffer cannot be used, rather on-chip addressable memory is used for data movement. The forward and backward loops can be made to operate in parallel when batching a number of system solves, using ping-pong buffers (also called double buffers). With this technique, dual port memory is partitioned into two parts, one being written while the other is read. Once writes (by the forward pass) and reads (backward pass) are completed, read and write halves are swapped. Note that the very first read must wait until the very first write has completed. Additionally, the technique also doubles the memory requirement compared to using the same memory portion for both read and write. The latencies for writing to the ping-pong buffer, firstly for a, b, c, d belonging to the first group of systems, then writing the resulting c^*, d^* in forward solve and finally writing u in backward solve, contribute to the latency term $3gN$ in (4.1). Here we assume, inputs a, b, c, d come from FIFO and output u is written back to FIFO. If inputs/outputs are read/written to on-chip memory instead, then (4.1) becomes $(1 + \lceil B/g \rceil) \times gN$.

The total on-chip memory required for a single Thomas solver interleaving g systems can be computed based on the need to store the a, b, c, d, c^*, d^* and u vectors, where each consumes $2gN$ words in the ping-pong buffers. The total $14gN$ requirement with dual port memory can be satisfied with $7\times$ dual port block RAMs (URAM/BRAM) each with a capacity of $2gN$. Additionally there is a need to store g values of the $(i-1)^{th}$ iteration separately, requiring 3 on-chip memories with a capacity of g words.

Data transfer from external memory to on-chip memory plays a crucial role in achieving high performance, especially for multi-dimensional solvers such as the 3D ADI heat diffusion application detailed later in this Chapter. If we consider a 3D application with systems sizes (N) of 256 in all three dimensions, then a solve along the x-dimension will have YZ (256×256 in this case) systems to be solved, each corresponding to an *x-line* system of size 256. Given the data is stored in consecutive memory locations along the x-lines, good memory throughput can be achieved. However to exploit the full memory bandwidth, a larger number of memory ports must be used. For the 512-bit memory ports, on the Alveo U280, it is sufficient to saturate the data-flow pipeline with a width of 256-bits at a 300MHz clock speed, which is our target frequency. This enables us to fetch data sufficient to feed 8 Thomas solvers in parallel. Such a configuration can be viewed as a *vectorized* Thomas solver. Additionally, the total YZ x-lines can be set up to be solved in groups (g) of 32. Here, the 1st Thomas solver datapath solves the 0th, 8th, 16th and so on x-lines, the 2nd solves 1st, 9th, 17th and so on x-lines, and so on. Batches of x-lines can be solved in such interleaved groups to saturate the dataflow pipeline to achieve higher throughput.

In the x-dimension, the reads from external memory bring in data stored in consecutive memory locations. However, the data fetched belongs to the same line (i.e. same system), thus we need to buffer 8 x-lines internally and carry out an 8×8 transpose to feed that to 8 different solvers (see Figure 4.1(a) for an illustration of the issue with a 4×4 transpose). For solving along the y-dimension, we fetch each XY plane to on-chip memory to avoid strided memory accesses and then read along the y-lines from the on-chip memory (see Figure 4.1(b)). Similarly for solving along the z-dimension, we read in x-lines (which are consecutive in memory) along the z dimension, fetching XZ planes, to on-chip memory. No transpose is required for y- and z-dimension solves as each element corresponds to

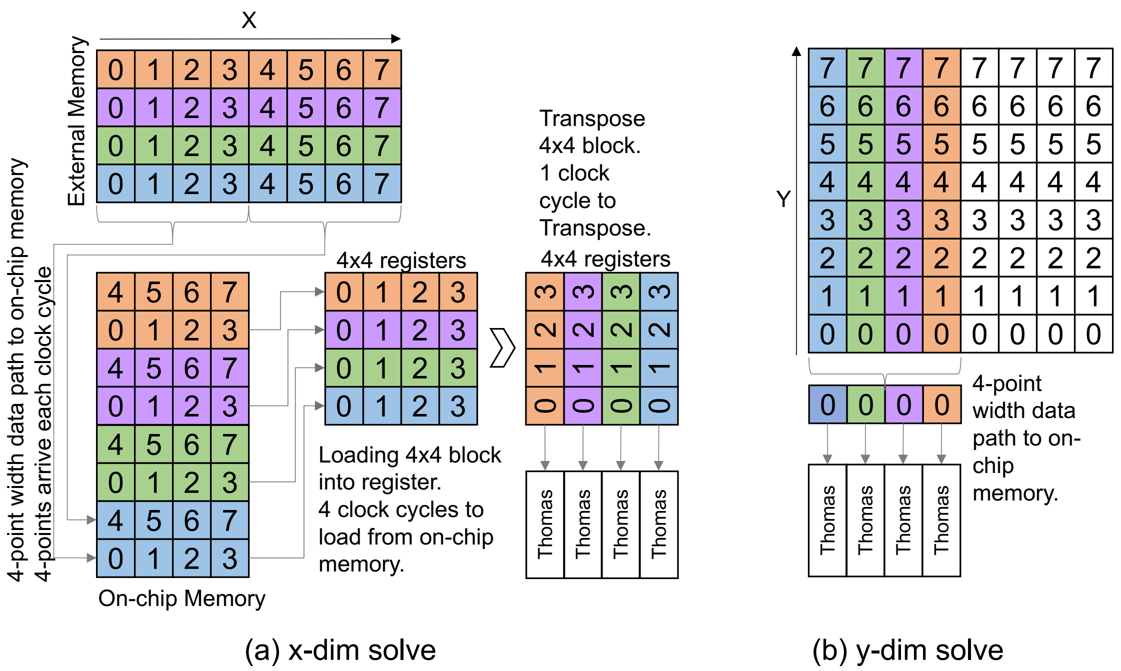

X
External Memory
4-point width data path to on-chip memory
4-points arrive each clock cycle
4x4 registers
Loading 4x4 block into register.
4 clock cycles to load from on-chip memory.
On-chip Memory
Transpose 4x4 block. 1 clock cycle to Transpose.
4x4 registers
Thomas
(a) x-dim solve
Y
4-point width data path to on-chip memory.
Thomas
(b) y-dim solve

Figure 4.1: Datapath for 4× (vectorized) x- and y-dim solves.

a different system. Utilizing the HBM available on modern FPGAs, the full vectorized Thomas solver, which can be viewed as a single compute unit (CU), can be instantiated a number of times to obtain further parallel performance. Specific designs for applications with multiple CUs are discussed in Section 4.2. For a 3D application, the x- and y-dimension solves can be effectively pipelined, storing the resulting XY planes in on-chip memory without writing to external memory. However the z-dimension solve requires reading from external memory. As such, 2D applications can be further optimized with unrolling. Again we discuss specific implementations with unrolling in Section 4.2.

4.1.2 Larger System Solvers

Interleaved solving of systems requires on-chip memory proportional to the system size, N, and number of groups g. As such, the maximum size of the system that can be solved is limited by the FPGA on-chip memory resources. We can split the tridiagonal system into subsystems (or *tiles*) of size M where each subsystem can be solved using a modified Thomas solver, where, after a forward and backward phase, each unknown is expressed in terms of two unknowns u_0 and u_{M-1}:

$$a_i u_0 + u_i + c_i u_{M-1} = d_i, \quad i = 1, 2, ..., M-2 \tag{4.4}$$

This results in a reduced tridiagonal system spread across each sub-domain as detailed by László *et al.* [42]). The unknowns at the beginning and end of each subsystem can be solved again using the Thomas algorithm, or indeed PCR. Finally, the result from the reduced system, is substituted back into the individual subsystems (see László et al. [42]

$$\left[\begin{array}{cccc|cccc} b_0 & c_0 & & & & & & \\ a_1 & b_1 & c_1 & & & & & \\ & a_2 & b_2 & c_2 & & & & \\ & & a_3 & b_3 & c_3 & & & \\ \hline & & & a_4 & b_4 & c_4 & & \\ & & & & a_5 & b_5 & c_5 & \\ & & & & & a_6 & b_6 & c_6 \\ & & & & & & a_7 & b_7 \end{array}\right] \left[\begin{array}{c} u_0 \\ u_1 \\ u_2 \\ u_3 \\ \hline u_4 \\ u_5 \\ u_6 \\ u_7 \end{array}\right] = \left[\begin{array}{c} d_0 \\ d_1 \\ d_2 \\ d_3 \\ \hline d_4 \\ d_5 \\ d_6 \\ d_7 \end{array}\right]$$

$$\left[\begin{array}{cccc|cccc} 1 & c_0 & & & & & & \\ a_1^* & 1 & c_1 & & & & & \\ a_2^* & & 1 & c_2 & & & & \\ a_3^* & & & 1 & c_3 & & & \\ \hline & & & a_4^* & 1 & c_4 & & \\ & & & & a_5^* & 1 & c_5 & \\ & & & & a_6^* & & 1 & c_6 \\ & & & & a_7^* & & & 1 \end{array}\right] \left[\begin{array}{c} u_0 \\ u_1 \\ u_2 \\ u_3 \\ \hline u_4 \\ u_5 \\ u_6 \\ u_7 \end{array}\right] = \left[\begin{array}{c} d_0^* \\ d_1^* \\ d_2^* \\ d_3^* \\ \hline d_4^* \\ d_5^* \\ d_6^* \\ d_7^* \end{array}\right]$$

$$\left[\begin{array}{cccc|cccc} 1 & & & c_0^* & & & & \\ a_1^* & 1 & & c_1^* & & & & \\ a_2^* & & 1 & c_2^* & & & & \\ a_3^* & & & 1 & c_3^* & & & \\ \hline & & & a_4^* & 1 & & & c_4^* \\ & & & & a_5^* & 1 & & c_5^* \\ & & & & a_6^* & & 1 & c_6^* \\ & & & & a_7^* & & & 1 \end{array}\right] \left[\begin{array}{c} u_0 \\ u_1 \\ u_2 \\ u_3 \\ \hline u_4 \\ u_5 \\ u_6 \\ u_7 \end{array}\right] = \left[\begin{array}{c} d_0^* \\ d_1^* \\ d_2^* \\ d_3^* \\ \hline d_4^* \\ d_5^* \\ d_6^* \\ d_7^* \end{array}\right]$$

Figure 4.2: Reduced system formation

which implements a Thomas-PCR solver for GPUs).

The *tiled*-Thomas-Thomas solver requires additional computation to solve the reduced system. To achieve higher performance, forward and backward phases over tiles can be interleaved. The reduced system size N_r is double the number of tiles. Solving the reduced system with Thomas requires $2gN_r$ clock cycles. This should not exceed the clock cycles taken by the forward and backward phases over the tiles. At the end of the backward phase, results (a^*, c^* and d^* as noted in [42]) are stored in a FIFO buffer while the reduced system for each tile is computed. Then the reduced system results can be substituted back to complete the solve. Using a FIFO maintains the dataflow pipeline without stalling.

Considering a system of size N, split into t tiles (note then $N_r = 2t$), assume we interleave g tiles using the Thomas-Thomas algorithm to solve a total of B systems. Then the total latency is given by (4.5):

$$(3 + \lceil Bt/g \rceil) \times \lceil N/t \rceil g + g_r \times (2t) \times 2 \qquad (4.5)$$

The second term is for the reduced solve. The g_r is similar to g, but it is equal to or larger than number of interleaved systems for the reduced solve. It is 32 for FP32 and

64 for FP64 on the U280. Similarly, based on the latency for solving the first phase of the algorithm on a tile, the number of systems to be interleaved is $\lceil 32/t \rceil$ for FP32 and $\lceil 64/t \rceil$ for FP64. For larger B, we can see that the latency tends to $Bt\lceil N/t \rceil$. Considering on-chip memory requirements the forward and backward phases of the modified Thomas can be shown to require $9 \times 2 \times g/t \times N$ words that can be satisfied by 9 on-chip memories setup as ping-pong buffers. Here we note that larger t lead to lower memory requirement. The reduced solve requires much less memory, $7 \times 2 \times 2t \times \lceil g/t \rceil$ in the form of 7 ping-pong buffers. Furthermore, a FIFO buffer would be required, of length equivalent to the maximum number of clock cycles spent on the reduced system, as we have to flush solved tiles from the backward phase.

The reduced system solve can also be implemented with the PCR algorithm resulting in the latency given in (4.6).

$$(3 + \lceil Bt/g \rceil) \times \lceil N/t \rceil g + (2t + l) \times log(2t) \tag{4.6}$$

Again for larger B, this tends to $Bt\lceil N/t \rceil$, however, there is a lower on-chip memory requirement of $(2t + l) \times log(2t)$ words for each of the 3 FIFO buffers, due to the lower latency for reduced system solve in PCR. Since dataflow design requires matching performance of solving tiles and the reduced system and as PCR is faster when solving reduced systems, the number of tiles can be increased even for smaller systems, further reducing the on-chip memory requirements for the first phase of the algorithm. As such we can expect the Thomas-PCR version to give better performance.

4.2 Performance

In this section we examine the achieved performance for the above FPGA design strategy. First, we briefly compare the performance of our library to a current state-of-the-art FPGA tridiagonal solver library from Xilinx [82] which is based on PCR, demonstrating the higher performance gains from a batched Thomas-based solver as predicted by the performance model developed in Section 4.1. Batching of systems is key to higher performance. Figure 4.3 presents the performance of 1D tridiagonal systems of size 128 and 1024, in FP32, solved using the Xilinx library (`xilinxlib-F1`) compared to our Thomas algorithm-based library (`tridsolvlib`) and tiled Thomas-PCR (`Tiled-tridsolvlib`) on a range of batch sizes. As predicted by the model, for larger batch sizes the Xilinx library performed significantly slower than the Thomas based solver. Adding further optimizations, such as inner loop unrolling and a FIFO data path to the Xilinx solver (`xilinxlib-F2`) only marginally improves performance, leaving an order of a magnitude performance gap. We also observe that the PCR-based `xilinxlib-F2` implementation consumes higher resources. `Tiled-tridsolvlib` breaks the systems into 32 tiles, and gives faster solve times compared to `tridsolvlib` for small batch sizes due to smaller tiles being solved in an interleaved manner.

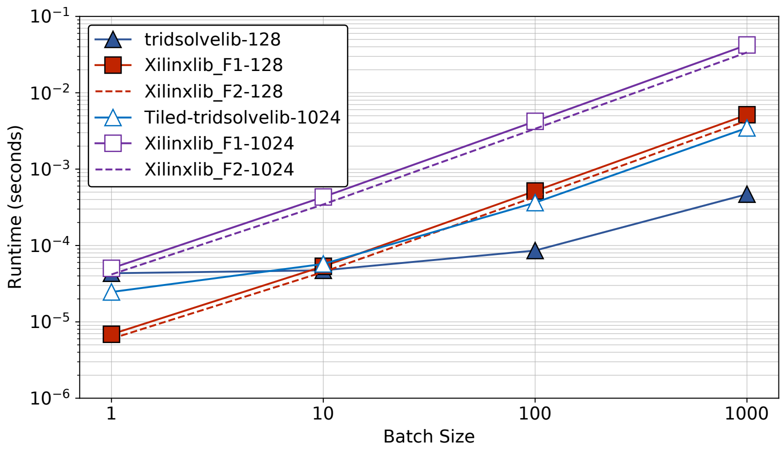


Figure 4.3: Proposed `tridsolvlib` vs `xilinxlib` (FP32) performance for system sizes of 128 and 1024.

In the remainder of this section we focus on using our FPGA design strategy, specifically applied to representative, non-trivial applications. We investigate both 2D and 3D applications, with both FP32 and FP64 precision. The performance models are used to determine initial design parameters and runtimes, which we compare to achieved runtimes on a Xilinx Alveo U280 (raw runtime values are available in Appendix section C.2). We use Vivado C++ due to ease of use for configurations and support of some C++ constructs compared to OpenCL. However, OpenCL could equally be used to implement the same design. Resources are estimated, with the aid of Vivado HLS tools. Finally, we compare performance on the FPGA to an Nvidia Tesla V100 GPU using the tridiagonal solver library, `tridsolver` implemented by László *et al.* [42, 77] using its batched version presented by Reguly *et al.* [64]. This GPU library has been shown [3] to provide matching or better performance than the two current batch tridiagonal solver functions in Nvidia's cuSPARSE library [14, 78] – `cusparse<t>gtsv2StridedBatch()` and `cusparse<t>gtsvInterleavedBatch()`. Our experiments also confirmed these results for the applications evaluated in this Chapter. Additionally it features direct support for creating multi-dimensional solvers, whereas `gtsvInterleavedBatch()` requires data layout transformations, for example in between doing an x-solve and a y-solve to implement multi-dimensional problems. The cuSPARSE `gtsv2StridedBatch()` library variant was observed to be slower. Thus we use `tridsolver` in our evaluation throughout this Chapter, but note that cuSPARSE libs would have equally provided the same insights when compared to the FPGA solvers on the Xilinx U280. Given that previous work has demonstrated GPUs to provide significantly better performance than multi-threaded CPUs [42], we do not compare with CPU implementations. Note that we only measure and present the time for the main iterative loop. As the applications carry out large numbers of iterations, the data copied to the device (on both devices via PCIe, incurring similar overheads), is used repeatedly. Therefore, the transfer overhead is amortized. Furthermore, with large multi-batch execution in real workloads, the initial transfer is

Table 4.1: Experimental systems specifications.

FPGA	Xilinx Alveo U280 [91]
DSP blocks	8490
BRAM/URAM	6.6MB (1487 blocks)/34.5MB (960 blocks)
HBM	8GB, 460GB/s, 32 channels
DDR4	32GB, 38.4GB/s, in 2 banks
Host	AMD Ryzen Threadripper PRO 3975WX (32 cores) 512GB RAM, Ubuntu 18.04.6 LTS
Design SW	Xilinx Vivado HLS, Vitis 2019.2
Run-Time	Xilinx XRT 202020.2.9.317
GPU	**Nvidia Tesla V100 PCIe [53]**
Global Mem.	16GB HBM2, 900GB/s
Host	Intel Xeon Gold 6252 @2.10GHz (48 cores) 256GB RAM, Ubuntu 18.04.3 LTS
Compilers, OS	nvcc CUDA 10.0.130, Debian 9.11

further hidden behind computation. Hence, data copy time from host to device (both on FPGA and GPU) are not included in our results.

Table 4.1 briefly details the specifications of the FPGA and the GPU systems (both hardware and software) used in our evaluation. The Nvidia V100 is based on 12nm technology while the Xilinx U280 is 16nm. The GPU also has a memory bandwidth of 900GB/s, nearly twice that of the U280's 460GB/s. Thus we selected the V100 as a fair but challenging competitor.

4.2.1 ADI Heat Diffusion Application

The first application is an Alternating Direction Implicit (ADI) based solve of the heat diffusion equation. The high-level algorithm of the application in 3D is detailed in Algo. 4.

Algorithm 4: `3D ADI Heat Application`

```
1: for $i = 0, i < n_{iter}, i++$ do
2:     Calculate RHS : $d = f_{7pt}(u), a = \frac{-1}{2}\gamma, b = \gamma, c = \frac{-1}{2}\gamma$
3:     Tridslv(x-dim), update $d$
4:     Tridslv(y-dim), update $d$
5:     Tridslv(z-dim), update $d$
6:     $u = u + d$
7: end for
```

The application consists of an iterative loop which starts by calculating the RHS values using a 7-point stencil, followed by calls to the tridiagonal solver for each of two or three dimensions, depending on the application. The updates from the tridiagonal solver, `Tridslvs` are accumulated to u before the next iteration. For the 3D ADI application, there are three calls to `Tridslv`. A GPU implementation has four kernels called by an iterative loop on the host. Fusing these kernels together does not improve performance as it requires global synchronization for data structure d and the memory accesses are along different directions of the 3D mesh, leading to poor cache utilization. The non-coalesced memory access pattern of `Tridslv(x-dim)` is a challenge for GPUs. László *et al.* [42] improved performance through shared memory and register based transposing.

An initial FPGA design implements the application as a single hardware unit given the data dependencies between the calls. This enables FPGA resource utilization to be maximized by implementing 6 CUs each having 8 Thomas solvers synthesized as a vectorized solver. The `RHS` calculation, which is a 3D explicit stencil loop was implemented using techniques similar to those in [33], as a separate module. The intermediate results between CUs and `RHS` module were written/read to/from external memory. The number of CUs is then limited by the available HBM ports but not by any other resource. An improvement on this initial design fuses the generation of a, b, c coefficients with the tridiagonal solver. This enables the required number of HBM ports to be reduced and synthesis of a maximum of 16 CUs. We opt for 12 CUs to reduce routing congestion which affects the maximum frequency achievable on the FPGA.

The x-dim and y-dim solves can be synthesized as separate modules, pipelining the X and Y dimension calculation without needing to buffer intermediate results in external memory. Essentially, XY planes are buffered in on-chip memory, but solvable mesh sizes are limited by BRAM/URAM usage. To also pipeline the z-dim solve the full mesh must be buffered on-chip which significantly limits the mesh size, hence we do not attempt it here. The pipelining reduces the bandwidth requirement by half compared to the previous design. The first module, `RHS + Tridslv(x-dim) + Tridslv(y-dim)` and second module, `Tridslv(z-dim)`, operate in parallel in a ping-pong fashion. This effectively increases the number of modules working in parallel to 24, considering the availability of HBM ports. The design now has a large pipeline start delay and is best utilized by batching large numbers of 3D meshes to obtain higher throughput. Xilinx dataflow design synthesis requires separate data structures for independent read and write operations. We introduce two data structures for accumulation in line 6 of Algo 4. But due to limited HBM ports, we must share a single HBM port between two data structures. This limits the dataflow per data structure from/to the HBM ports as well as the size of data structure, given a single HBM bank has a capacity of 256MB. This final design gave the best performance in our evaluations.

The component model in (4.1) can be combined with the delays due to buffering (ping-pong buffers for the 8×8 transpose, row-to-col, rows-to-8×8-block data flow and window buffers for stencil computations) to obtain an application performance model. These delays are determined by the clock cycles needed to fill the buffers in order to start outputting the first result. Thus the full pipeline latency for the 3D ADI application

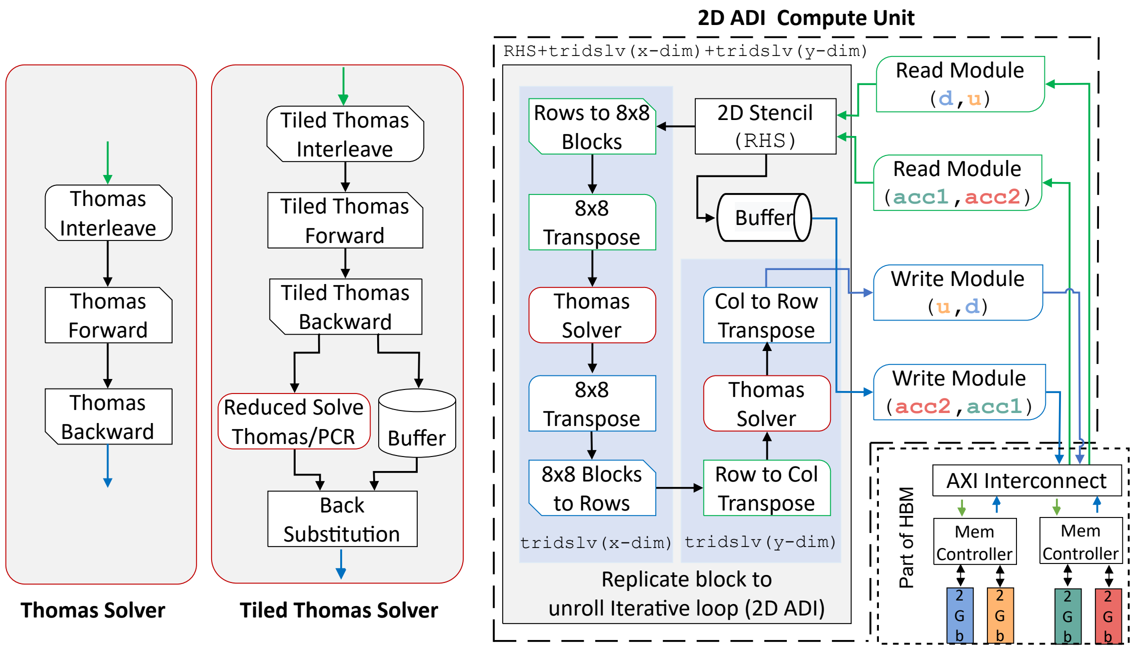


Figure 4.4: 2D ADI application datapath constructed from solver components.

is (4.7):

$$L_{adi,3D} = n_{iter} \times MAX(L_{rhs+xy}, L_z) \tag{4.7}$$

$$L_{rhs+xy} = (xy/V) + (2Vx/V + 3gx) + (2xy/V + 3gy) + \lceil B/2N_{CU} \rceil (xyz/V) \tag{4.8}$$

$$L_z = (2xz/V + 3gz) + \lceil B/2N_{CU} \rceil (xyz/V) \tag{4.9}$$

Here, x, y and z are the sizes of systems in each dimension, N_{CU} is the number of CUs implemented on the FPGA and B is the total number of 3D meshes, i.e the number of batches. The terms in (4.8) account for the 3D stencil computation in `RHS`, `Tridslv(x-dim)` including latency to transpose the x-lines, `Tridslv(y-dim)` including the reading/writing y-lines from the buffered x-lines, and the latency to process B meshes using N_{CU} CUs respectively.

We take the maximum in (4.7) because the two modules need to be synchronized, as they swap their read and write locations after processing $B/2$ meshes. The vectorization factor V is 8 for our design and g is 32 for FP32 and 64 for FP64. A minor consideration for obtaining improved predictions from the above model is when the number of points per clock cycle arriving to the vectorized solvers is different to V due to memory bandwidth. For example if we use a single HBM port to read two data structures and if we use a 256-bit data path, a lower number of points p will enter the datapath than V. Then, replacing V by p is more accurate.

A similar design can be developed for the 2D ADI application, but now the functions

Table 4.2: 2D ADI Heat Diffusion App. : Achieved Bandwidth (GB/s) and Energy (J) on the FPGA (F) and GPU(G).

2D FP32 (120 iterations, $f_U = 3$)										
Batch Size	1500					3000				
	Bandwidth			Energy		Bandwidth			Energy	
Mesh	F	Gx	Gy	F	G	F	Gx	Gy	F	G
32^2	501	375	276	1	5	563	435	377	2	9
64^2	524	428	449	3	16	556	447	512	6	29
128^2	602	416	539	12	60	620	418	554	23	115

2D FP64 (120 iterations, $f_U = 2$)										
Batch Size	1500					3000				
	Bandwidth			Energy		Bandwidth			Energy	
Mesh	F	Gx	Gy	F	G	F	Gx	Gy	F	G
32^2	360	396	501	2	7	395	441	535	4	14
64^2	380	401	492	9	30	399	417	506	18	61
128^2	411	286	512	34	141	422	281	548	67	284

in the iterative loop `RHS`, `Tridslv(x-dim)` and `Tridslv(y-dim)` can all be pipelined. This makes it possible to unroll the iterative loop by some factor f_U. Note that the variable u is incremented each iteration (line 6 of Algo. 4), where the previous value of u must be input at the end of each unrolled iteration to carry out this increment. However the `RHS` of each iteration also consumes u and thus we use a delay-buffer (similar to ones used in StencilFlow [15]) implemented as an HBM FIFO to feed the previous values of u to the increment stage on line 6. Implementation of an HBM FIFO with a data access dependency distance based on the data structures allocated on specific HBM banks makes global memory synchronization possible in the dataflow pipeline without additional HBM throughput cost. Unrolling the iterative loop reduces the total number of data structures in external memory. Hence we are able to assign dedicated ports for each data structure which enables better dataflow throughput. The overall structure of the 2D design, combining component modules is illustrated in Figure 4.4. A similar illustration can be conceived for the 3D ADI application, which we do not show here. The performance model for the 2D application is given in (4.10).

$$L_{adi,2D} = (n_{iter}/f_U) \times L_{rhs+xy} \tag{4.10}$$

$$\begin{aligned} L_{rhs+xy} =& f_U \times [(x/V) + (2Vx/v + 3gx) + (2xy/V + 3gy)] + \\ & \lceil B/N_{CU} \rceil (xy/V) \end{aligned} \tag{4.11}$$

Pipeline latency increases with the unroll factor f_U, but for large B it results in a higher overall speedup. The size of the FIFO delay buffer is equivalent to the total delay

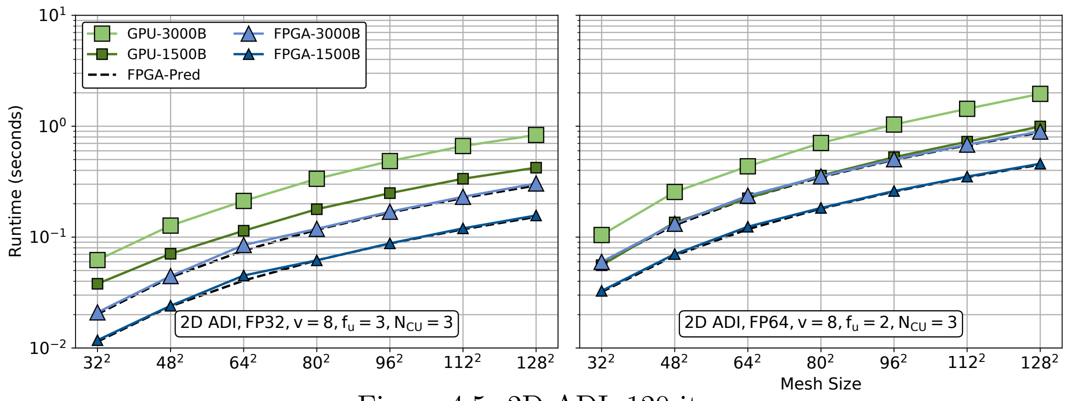


Figure 4.5: 2D ADI: 120 iter

of RHS, `Tridslv(x-dim)`, and `Tridslv(y-dim)` : $x/v + 2vx/v + 3gx + 3gy + 2xy/v$.

Figure 4.5 details the performance of the 2D ADI Heat diffusion application implemented in both FP32 and FP64 on the FPGA and compares it to execution on the GPU. The design parameters for each are noted in the graphs. Operating frequencies are 292MHz and 288MHz for FP32 and FP64 respectively. These improved post implementation frequencies were possible due to multiple compute units with careful SLR placement and HBM bank assignment constraints, manually flattened loops with arbitrary word length counters, and an optimally pipelined and vectorized design. In both FP32 and FP64 cases the coefficients a, b and c are internally generated, on the FPGA. This means that only u is read. Performance results demonstrate the FPGA outperforming the GPU particularly for runs with large batch sizes.

We see that the performance model accuracy is over 85% with large batched predictions being more accurate at over 90%. The prediction errors in the models are due to omitting a number of minor latencies for simplicity. While the models accounts for only the latency in loops, real synthesized circuit on the FPGA will have additional stages to complete before a loop. These include calculating the loop invariants and initial values and state transition to function calls. For modeling loops, we do not account for the hardware pipeline latency - i.e. the clock cycles required between FIFO read and write and arithmetic hardware pipeline. However, these can be obtained from the HLS kernel schedule viewer to refine and improve predictions. We also do not account for latency incurred on the first external memory transfer. All of the above latencies are less than a few hundred clock cycles and become insignificant when considering total runtimes of larger mesh or batch sizes as can be seen from the above results.

Inspecting the effective bandwidth on each device as detailed in the two sub-tables in Table 4.2 provides insights into the superior performance of the FPGA. The bandwidth is computed by counting the total number of bytes transferred during the execution of each call in Alg. 4, looking at the mesh data accessed and dividing it by the total time taken by each call. On the GPU, we have detailed the achieved bandwidth of the x- (Gx) and y-dim (Gy) solves. On the FPGA we show the full bandwidth achieved in the pipeline. The x-dim bandwidth on the GPU is significantly worse due to the block transpose operations.

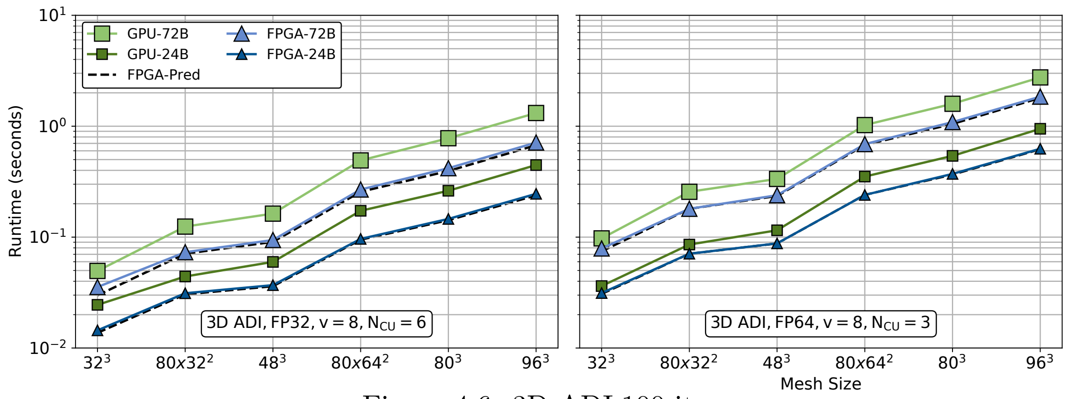


Figure 4.6: 3D ADI:100 iter

Such lower bandwidths are also confirmed by László *et al.* [42]. We additionally confirmed the same performance when using cuSPARSE's `cusparse<t>gtsv2StridedBatch()` library function for the x-solve. The higher performance of the FPGA can be attributed to the unrolling of the iterative loop, keeping intermediate results in fast on-chip memories, thus allowing higher bandwidth utilization for the data path and the internal generation of coefficients. The GPU tridiagonal solver library does not support internal coefficient generation. Thus, the application writes a, b, c and u to global memory after `RHS` and intermediate results also written/read between the two `Tridslv` calls, whereas on the FPGA these stay on-chip. Even with modifications to the GPU library to generate coefficients internally which would improve GPU performance, we believe the FPGA results point to a very competitive solution, particularly when batching large meshes that can fit within the resource constraints of the FPGA, for this application.

The first two sub-tables in Table 4.2 also detail the energy consumption of the 2D runs. The `xbutil` utility was used to measure power during FPGA execution, while `nvidia-smi` was used for the GPU. The FPGA on average consumed 75W while the GPU power draw ranged from 50W to 250W. Results indicate that the FPGA energy consumption is approximately 5–6× lower for this 2D problem.

Figure 4.6 and the two sub-tables in Table 4.3 detail the performance of the 3D ADI heat diffusion application in FP32 and FP64 respectively. Again we see performance trends similar to the 2D case, however we were only able to run smaller batch sizes due to HBM memory limitations for 3D meshes. On the GPU, again, we observe good achieved bandwidth. On the FPGA the achieved bandwidth is poorer due to no unrolling of the iterative loop as done in the 2D case, where there are 3 CUs each unrolled by a factor of 3. The sharing of HBM ports as described in the design of this application limits the data flow per data structure further reducing achieved bandwidth. The energy consumption of the FPGA is 3–4× lower than the GPU's.

A Thomas-Thomas based implementation for the 2D ADI-Heat application for larger

Table 4.3: 3D ADI Heat Diffusion App. : Achieved Bandwidth (GB/s) and Energy (J) on the FPGA (F) and GPU(G).

3D FP32 (100 iterations)

Batch Size	24						72					
	Bandwidth				Energy		Bandwidth				Energy	
Mesh	F	Gx	Gy	Gz	F	G	F	Gx	Gy	Gz	F	G
$32 \times 32 \times 32$	218	380	229	283	1	3	266	449	390	537	3	8
$48 \times 48 \times 48$	288	426	354	477	3	9	338	459	408	553	7	25
$96 \times 96 \times 96$	346	401	401	568	18	65	358	415	419	563	53	197

3D FP64 (100 iterations)

Batch Size	24						72					
	Bandwidth				Energy		Bandwidth				Energy	
Mesh	F	Gx	Gy	Gz	F	G	F	Gx	Gy	Gz	F	G
$32 \times 32 \times 32$	201	405	365	445	2	4	239	439	424	527	6	13
$48 \times 48 \times 48$	242	388	407	536	7	16	267	408	421	554	18	48
$96 \times 96 \times 96$	271	324	412	550	47	135	276	338	436	565	139	399

meshes can be modeled using (4.12):

$$L_{adi,2D,tiled} = n_{iter} \times (L_{rhs+x} + L_y) \tag{4.12}$$

$$L_{rhs+x} = x/V + 2Vx/V + 3gx/t_1 + 4gt_1 + Bxy/V \tag{4.13}$$

$$L_y = 2yT_x/V + 3gy/t_2 + 4gt_2 + Bxy/V \tag{4.14}$$

In this case, `RHS` and x-solve can be pipelined but y-solve cannot as we are computing "tiles" along the y-dim lines, a large amount of on-chip memory would be required to transpose the mesh. The explicit stencil computation in RHS does not require tiling as we are not processing very large meshes. If the tile sizes for the Thomas-Thomas solvers are selected to be t_1 and t_2 then the reduced system sizes will be $2t_1$ and $2t_2$. Equation (4.13) accounts for the latency for `RHS` with x-dimension solve where the terms correspond to the latencies of the stencil, the data path, modified Thomas solve, and the reduced solve. Similarly (4.14) gives the y-dimension solve latency. Note that here we have used T_x (this is different to $t1$) as the tile size for the y-dim data path where we buffer $T_x \times y$ sized planes. Note also that we have selected the number of interleaved systems and interleaved reduced systems to be equal (i.e. $g = g_r$ in relation to (4.5)). The final term in (4.13) and (4.14) are the latencies for processing a batch of B systems. Replacing the reduced system solve with the PCR algorithm is also possible where the $4gt_1$ and $4gt_2$ terms in (4.13) and (4.14) then become $log(2t_1) \times (2t_1 + l)$ and $log(2t_2) \times (2t_2 + l)$. Here, l is circuit pipeline latency as discussed in Section 4.1.

Figure 4.7 presents the performance of the 2D ADI heat diffusion application on

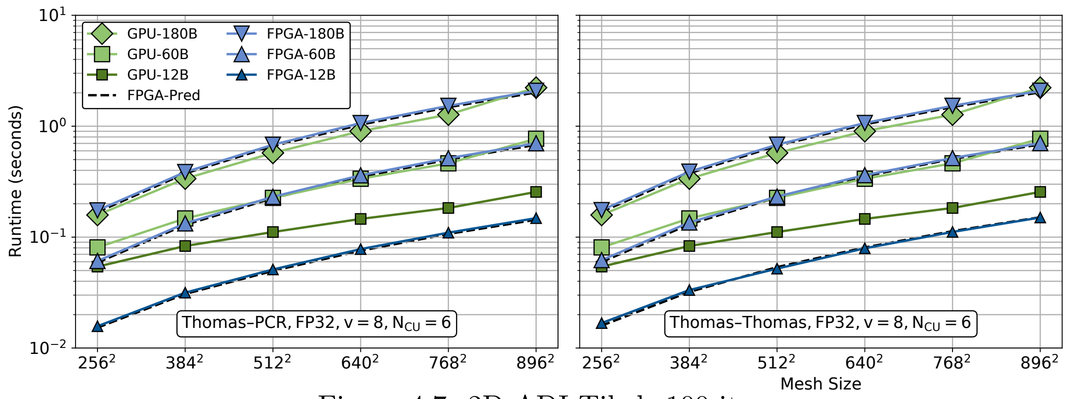


Figure 4.7: 2D ADI-Tiled: 100 iter

Table 4.4: ADI Heat Diffusion App (2D FP32) – Large meshes, Thomas-PCR: 100 iterations, Bandwidth (GB/s), Energy (J) on the FPGA (F) and GPU (G).

Batch Size	60					180				
	Bandwidth			Energy		Bandwidth			Energy	
Mesh	F	Gx	Gy	F	G	F	Gx	Gy	F	G
256^2	692	213	238	5	8	217	766	437	13	21
512^2	218	768	379	17	28	222	797	534	51	75
896^2	220	345	495	53	104	222	342	562	156	312

large meshes solved using Thomas-PCR and Thomas-Thomas hybrid implementations. Again we compare with the same mesh sizes solved on the GPU. Due to the `RHS` and `Tridslv(x-dim)` being pipelined together, the FPGA achieves better HBM bandwidth utilization. The GPU also achieves good bandwidth utilization where it reaches bandwidth levels similar to batched smaller meshes (see Table 4.4 for for Thomas-PCR; Thomas-Thomas gave similar results). The FPGA can be seen to be 2–3× more energy efficient than the GPU for the largest mesh sizes.

4.2.2 Stochastic Local Volatility

The second application we evaluate comes from computational finance. It implements a Stochastic-Local Volatility (SLV) model, which describe asset price processes, particularly foreign exchange rates [74]. A batched GPU implementation based on a second order finite-difference scheme was developed for this problem using the OPS DSL by Reguly *et al.* [64]. It is a 2D application implemented in FP64 precision. Its high-level algorithm is detailed in Algo. 5.

The application implements a Hundsdorfer-Verwer (HV) method, (also based on the ADI method) for time integration. The Rannacher smoothing available in the original application has been switched off in our evaluation. The `hv_pred*` and `hv_matrices` are explicit loops each using 10 point stencils, requiring a window buffer implementation [33] for data reuse. The 9 kernels in Algo. 5 were implemented as separate hardware modules,

Algorithm 5: `2D Heston SLV Backward`

```
1: for $i = 0, i < n_{iter}, i++$ do
2:     hv_pred0(), hv_matrices()
3:     Tridslv(x-dim)
4:     hv_pred1(), Tridslv(y-dim)
5:     hv_pred2(), Tridslv(x-dim)
6:     hv_pred3(), Tridslv(y-dim)
7: end for
```

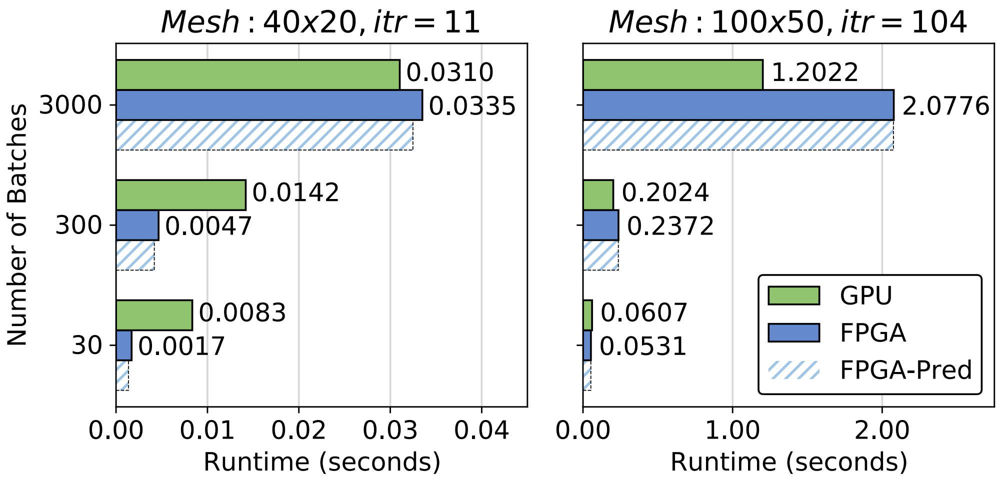


Figure 4.8: SLV application performance.

pipelining the computation within the iterative loop. `hv_matrices` generates a number of 2D coefficients `AX,BX,CX,AV,BV,CV` and 1D coefficient `EV` for the `Tridslvs`. Coefficients `AX,BX,CX` then needs to be input to (consumed by) `Tridslv(x-dim)` kernels and coefficients `AV,BV,CV` and `EV` to `Tridslv(y-dim)` kernels. A GPU implementation consists of these nine kernels, moving data through global memory. Again, it is not possible to fuse kernels to reduce global memory accesses for this bandwidth limited application. The large number of kernels inside the iterative loop incur significant kernel call overhead and data movement through the fixed data path increases latency on the GPU for processing meshes with smaller batch sizes, leading to poor bandwidth utilization.

On FPGA, generated coefficients are consumed at different stages of the pipeline. However other inputs to the `Tridslv` calls come through the computation of this multi-stage pipeline. Therefore large FIFO delay buffers are required to keep synchronization (i.e. avoid pipeline stalling). As such we opt to regenerate the above coefficients at separate stages, essentially duplicating the circuitry. This results in the generation of coefficients `AX,BX,CX`, for the `Tridslv(x-dim)`, being fused to `hv_pred0()` and `hv_pred2()` and the generating of coefficients `AV,BV,CV,EV`, for `Tridslv(y-dim)`, being fused to `hv_pred1()` and `hv_pred3()`. This results in a total of 8 hardware modules, requiring significantly smaller delay buffers than if we implemented the original set of kernels. The performance

Table 4.5: SLV Application, Bandwidth (GB/s) and Energy (J).

	Bandwidth			Energy	
Batch	FPGA	GPU-x	GPU-y	FPGA	GPU
40×20 mesh: 11 iterations					
30	55.24	3.04	28.01	0.13	0.45
300	202.31	16.48	176.51	0.35	1.02
3000	281.06	123.84	327.65	2.51	4.75
100×50 mesh: 104 iterations					
30	124.63	51.28	109.65	3.98	3.76
300	278.87	235.22	238.34	17.79	22.26
3000	318.36	421.77	429.21	155.82	216.40

model for SLV is given in (4.15):

$$\begin{aligned} L_{slv} = n_{iter}[&4 \times (2x) + 2 \times (3gx) + \\ &2 \times (3gy + 2xy) + \lceil B/N_{CU} \rceil xy] \end{aligned} \tag{4.15}$$

Here g is 64 as SLV uses FP64. The first term is the combined input/output latency for the four explicit stencil computations in `hv_pred*`. The second and third terms account for the `Tridslv (x-dim)` and `Tridslv(y-dim)` calls respectively, including the read or write y-lines from the buffered x-lines. The final term is the latency for processing a batch size of B, 2D meshes. The number of CUs, N_{CU} for SLV on the FPGA is 3, given the considerably larger amount of DSP and memory resources required for the application, particularly due to its use of FP64 precision. The FIFO delay-buffer size calculation was aided by the Xilinx HLS tools where the exact datapath pipeline latency was estimated to obtain buffer sizes adequate for an implementation.

The motivation for batched solves of multi-dimensional tridiagonal systems primarily comes from financial computing where, for example, computing prices of financial options and managing risk by hedging options leads to the need to solve Algo. 5 type applications with different sets of coefficients [64]. Additionally carrying out extensive speculative scenarios required by regulators under various market conditions to evaluate a bank's exposure means that there are large number of options in the order of thousands to hundreds of thousands to be computed every day. Such workloads would entail large numbers of roughly identical PDE problems to be solved which are well suited to be batched together.

Figure 4.8 and Table 4.5 detail the runtime, bandwidth, and energy performance of the SLV application implementation. Only two specific mesh sizes were available from the authors of the original code [64], each was batched up to 3000 batches of 2D meshes for this evaluation. The application is significantly more complex given the additional explicit stencil loops as well as the tridiagonal solvers. The runtimes here were obtained with the FPGA operating at 253MHz. As can be seen from the figures, the FPGA in some cases is faster than the V100 GPU, but for the largest batch sizes we attempted here, it is 8%-70% slower than the GPU. However the FPGA solution is over 30% more energy efficient for large batch solves compared to the GPU. The achieved bandwidth on the FPGA is approximately at the same level as the 2D ADI FP64 version. Runtime predictions from the model were also observed to be over 90% accurate for all cases.

4.3 Discussion

The experiments in Section 4.2.1 show better performance on the Xilinx Alveo U280 FPGA compared to the Nvidia V100 GPU for ADI 2D and ADI 3D applications in both FP32 and FP64 formats. Key optimizations possible on the FPGA, such as pipelining and fusing coefficient generation with tridiagonal solvers leads to this performance gain. These optimizations helped to achieve higher effective bandwidth on the FPGA although U280 HBM's maximum theoretical bandwidth (460GB/s) is close to half of the V100 HBM (900 GB/s). Additionally, lower FPGA resource consumption due to these optimization makes it possible to scale to multiple compute units on the Alveo U280. Implementation of an 8×8 transpose on the FPGA enables higher throughput for `Tridslv(x-dim)` making memory accesses coalesced, while the GPU implementation using shared memory based transpose and Tridslv to address non-coalesced accesses suffers significant performance loss. In Section 4.2.2, the FPGA demonstrates competitive performance with the GPU for the SLV application. However, the computationally intensive complex coefficient calculation using 10-point stencils makes it hard to fuse with the Thomas solver and results in higher FPGA resource usage, limiting the number of implementable compute units. Due to this, the GPU performs better than the FPGA for the SLV application on larger meshes. Future FPGAs with more DSP blocks or floating point primitives will provide better performance than the Xilinx Alveo U280. However, SLV with smaller meshes/-batches is better matched to the FPGA due to the low latency FPGA data movement as well as lower kernel call overhead as the iterative loop is implemented within the FPGA

4.4 Concluding Remarks

We have developed a new FPGA-based tridiagonal solver library aimed at solving multiple multi-dimension tridiagonal systems on FPGAs. Key new features of the library include dataflow techniques and optimizations for gaining high throughput, through batching multiple system solves, replication of compute units, and utilization of High Bandwidth Memory on modern FPGAs. The Thomas algorithm was shown to be effective, even with its loop carried dependencies, due to its simplicity and lower resource consumption. This somewhat subverts the conventional expectation of the more parallel PCR or SPIKE algorithms being better suited for high performance on parallel architectures. Our library significantly outperforms the Xilinx tridiagonal library that uses the PCR algorithm, for larger batch sizes. However, for larger mesh sizes a hybrid Thomas-PCR or Thomas-Thomas solution was required to overcome the limitations of on-chip memory and demonstrated considerable performance with batched configurations.

Two representative applications, (1) a heat diffusion problem based on the ADI method and (2) a stochastic local volatility (SLV) model from the financial computing domain, that rely on the solution of multi-dimensional tridiagonal systems were implemented using the new library on a Xilinx Alveo U280 FPGA. As part of the design process an analytical performance model was developed to estimate runtime performance of the FPGA designs and assist in design space evaluations. The FPGA performance was compared to optimized solutions of the same applications on a modern Nvidia Tesla V100 GPU, showing competitive performance, sometimes even surpassing the performance on the GPU. This was due to designs creating longer pipelines keeping intermediate results on fast FPGA on-chip memory.

Even when runtime is inferior to the GPU, significant energy savings, over 30% for the most complex application (SLV) with large batch sizes, were observed. Considering the motivating real-world scenario for such an application from the financial computing domain, such energy savings point to a significant operational cost benefit. The analytical performance model provides over 85% accuracy illustrating its significant utility in developing profitable FPGA designs. The results showcase a key class of applications and their characteristics where the FPGA is able to provide competitive performance on-par with GPUs, with the added benefit of large energy savings. The techniques and optimizations required to achieve high performance on FPGAs, as demonstrated in this work, provide key insights into the feasibility and profitability of using FPGAs in high-performance computing workloads.

The FPGA library, the 2D/3D ADI heat diffusion application, and the optimized GPU source code developed in this work are available as open-source software at [36]. The library and workflow are tested again on Xilinx Alveo U50 FPGA (see Appendix B) as well, which further supports the conclusions in this Chapter. The next Chapter explores the use of FPGA hardware from Intel, the other major FPGA device vendor.

Chapter 5

FPGA Designs with SYCL

This Chapter explores the design and development of structured mesh based solvers using the SYCL programming model on the Intel FPGA hardware, based on previously developed workflow (Chapter 3-4). Both explicit and implicit classes of applications are targeted : (1) stencil applications based on explicit numerical methods and (2) multi-dimensional tridiagonal solvers based on implicit methods. Special optimisation & techniques along with finer predictive models for SYCL Programming model and intel FPGAs to explore design space and get optimised design is presented in this Chapter. Performance of synthesized designs, using the above techniques, for two non-trivial applications on an Intel PAC D5005 FPGA card is benchmarked. Results are compared to the performance of optimized parallel implementations of the same applications on a Nvidia V100 GPU. Observed runtime results indicate the FPGA providing comparable or improved performance to the V100 GPU. However, more importantly the FPGA solutions consume 59%–76% less energy for their largest configurations. Our performance model predicts the runtime of designs with high accuracy with less than 5% error for all cases tested, demonstrating significant utility for design space exploration on Intel FPGAs. With these tools and techniques, this Chapter discusses special optimisations and techniques using SYCL to target intel FPGAs compared to C++ for Vivado for Xilinx FPGAs and how to code designs using SYCL, and the resulting performance.

5.1 Intel FPGAs and SYCL

Similar to Xilinx FPGAs which we have utilized in Chapters 3-4, Intel FPGA devices consist of basic circuit elements such as configurable logic, known as Adaptive Logic Moduless (ALMs) in Intel devices, that include LUTs and registers; specialized blocks such as random-access-memory blocks (640-bit MLABs and 20K bits M20K in Intel devices); and Digital Signal Processings (DSPs) blocks. These are interconnected via a rich routing fabric providing large bandwidth between elements. In addition to on chip memories, Intel FPGA boards comes with larger DDR4 memories and some modern FPGA boards comes with HBM2 memory. In contrast to Xilinx FPGAs, Intel FPGAs include DSP blocks

```
using namespace sycl;
void stencil_WI( queue &q,
              buffer<float,2> b_data_in,
              buffer<float,2> b_data_out,
              int size0, int size1,
              int block0, int block1){
  q.submit([&] (handler& h){
    accessor in(b_data_in, h);
    accessor out(b_data_out, h);

    range<2> local_range(block0, block1);
    range<2> global_range(size0, size1);

    h.parallel_for<class stencil_WI>
    (nd_range<2>(local_range, global_range),
    [=] (nd_item<2> point){
      int y = point.get_global_id(0);
      int x = point.get_global_id(1);
      if(x > 0 && y > 0 && x < size0-1 && y < size1-1){
        float r = (in[y-1][x] + in[y+1][x])*0.125f +
          in[y][x]*0.5f;
        out[y][x] = r;
      }
  });});
}
```

Listing 5: `NDRange` based stencil computation.

which support low latency single precision (FP32) `ADD`,`SUB`,`MUL` and `ACCU` operations. This helps modern Intel FPGAs to achieve higher computational throughput. As a given circuit design grows and begins to occupy a larger portion of the FPGA, routing (i.e. connecting all the circuit elements together) becomes more challenging, and can reduce the achievable clock frequency and hence overall performance. In this aspect, Intel's Hyperflex technology [28] gives more flexibility in routing which helps achieve better clock frequency.

The recently introduced Data Parallel C++ (DPC++) programming model[1] based on the SYCL programming model to program FPGAs, follows OpenCL. Here, the portions of the program to be executed on an accelerator device are called *kernels*. SYCL's accelerator model consists of a number of compute units, each made of processing elements (similar to SMs on a GPU). An instance of a kernel executed on a processing element is called a *work-item* (equivalent to *threads* in the Compute Unified Device Architectures (CUDAs) programming model) and an instance of a work-item is identified using an index *id* in a global index space. Work-items are organized into groups called *work-groups* (*thread-blocks* in CUDA) and each work-item inside the group will have a local-id. Work-items in a work-group are executed concurrently on processing elements of the compute unit where the index space is specified using SYCL's N-dimensional range model. Kernels based on

[1]Terms DPC++ and SYCL are used interchangeably in this Chapter due to the use of Intel target hardware.

```
using namespace sycl;
void stencil_ST(queue &q,
                buffer<float,2> &b_data_in,
                buffer<float,2> &b_data_out,
                int size0, int size1){
  q.submit([&] (handler& h){
    accessor  in(b_data_in, h);
    accessor  out(b_data_out, h);
    h.single_task<class stencil_ST> ([=] (){
      /* optimisations - see Section 2 ... */
      float window1[1024];
      float window2[1024];
      [[intel::loop_coalesce(2)]]
      /* +1 due to one row delay through window buffer */
      for(int y = 0; y < size1+1; y++){
        for(int x = 0; x < size0; x++){
          float s_12;
          if(y < size1) s_12 = in[y][x];
          float s_11 = window1[x];
          float s_10 = window2[x];
          window1[x] = s_12;
          window2[x] = s_11;
          float r = (s_10 + s_12)*0.125f + s_11*0.5f;
          if(x > 0 && y > 0 &&
             x < size0-1 && y < size1){
             out[y-1][x] = r;
          }
        }
      }
  });});
}
```

Listing 6: `single_task` based stencil computation.

this index space are called `NDRange` kernels (see Listing 5).

`NDRange` kernels therefore essentially follow a Single Instruction Multi Thread (SIMT) execution model where on a GPU, multiple kernels are called, with the system scheduling them to be executed on the available Streaming Multi-Processors (SMs). Such an execution can be done for FPGAs as well, but performance becomes severely limited due to FPGA global memory bandwidth (about 19–76GB/s with DDR4 or ≈ 460GB/s with HBM2, compared to over 900GB/s on modern GPUs) when having to write the results of one kernel back to global memory before calling the next kernel and the new kernel having to read all the necessary data back from global memory. However, on-chip memory bandwidth on FPGAs exceeds tens of TB/s and therefore feeding the results from one kernel to the next in a pipeline, provides significant opportunities for performance gains. This is achieved by a *single task* kernel, by attempting to create longer and longer computational pipelines essentially following a Multiple Instruction Multiple Data (MISD) model. A sequence of kernels within nested loops then lead to flattening the loop nests and fusing the loops (see discussion in Section 5.2), of course within the resource limits of the FPGA for implementing the computation. With SYCL, such a single-task kernel can be codified as in Listing-6.

In comparison to OpenCL, which is used by most of the previous work on Intel FPGAs [84, 83, 96, 95, 32, 49, 48], SYCL provides a higher level of abstraction, including removing much of the fixed "boiler-plate" code segments required to setup the device. Additionally, kernel arguments need not be set explicitly and data is automatically moved from host to device through `sycl::buffers`. Device memory is released when these buffers run out of scope. Essentially, the SYCL run-time makes sure that data is available on device/host before the execution of the kernel/host part of the program. The runtime additionally analyzes data dependencies where a kernel consuming dependent data will not be scheduled for execution the until completion of kernels that produce that data. This introduces the limit of only one kernel being able to write to a data structure at a time. An example of issues due to this limitation includes the challenge of moving the time-marching loop in an explicit stencil computation to the FPGA. This and other key designs for synthesizing the two classes of applications on Intel FPGAs using SYCL are discussed in the next two sections.

5.2 Stencil Solvers

Chapter 3 developed a generalized workflow for synthesizing stencil applications on Xilinx FPGAs. In this section we extend the workflow to Intel FPGAs with SYCL. The full FPGA designs implemented with SYCL can be found in [37].

Nested Loop Unrolling

As we alluded to previously, multi-dimensional nested loops should therefore be flattened to a 1D loop either manually or by using HLS directives in-order to avoid the clocks required to flush data from inner loop's circuit pipeline. With SYCL (specifically with the `Intel® oneAPI DPC++/C++` compiler, Intel's implementation of a SYCL compiler), we can easily achieve this by using the `loop_coalesce` attribute specified as a pragma on the nested loop.

Vectorisation (Cell Parallel method)

Replicating the circuitry for the elemental computation can also be done, provided (1) there are no data dependencies between the nested loop iterations and (2) there are enough resources available for synthesis on the FPGA (e.g. DSP units, FP cores etc). For stencil applications there are no such dependencies. This will lead to multiple pipelines (number limited by resources) operating in parallel, similar in operation to a vector operation on CPUs. This technique is also known as the *cell-parallel* method [84, 83], where if you visualize the stencil computation implemented with a nested loop as a loop over a regular multi-dimensional rectangle of mesh points/cells, this method will compute

```
/* Data type for wider data path */
struct dPath16 {[[intel::fpga_register]]float data[16];};

for(int i = 0; i < total_itr; i++){
 struct dPath16 s_1_0, s_1_1, s_1_2, vec_wr;
 /* other declarations, index calculation, window buffer*/
 #pragma unroll VFACTOR
 for(int v = 0; v < VFACTOR; v++){
  int i_ind = i *VFACTOR + v;
  float val = (s_1_0.data[v]+s_1_2.data[v])*0.125f+ \
             s_1_1.data[v]*0.5f;
  bool cond = (i_ind>0 && i_ind<size0-1 && j>1 && j<size1);
  vec_wr.data[v]= cond ? val : s_1_1.data[v];
  }
  /* writing results to pipe */
}
```

Listing 7: Vectored stencil computation.

multiple "cells" in parallel. For SYCL synthesis, this requires reading a wider block of memory, we prefer to program with a struct based data type as in Listing 7.

Window Buffers

Next, the design flow from Chapter 3 specifies the need to stream data from/to external (DDR4) and near-chip (HBM2) memories to/from on-chip MLABs/M20Ks to feed the computational pipeline efficiently. A perfect data reuse path can be created by (1) using a First-In-First-Out (FIFO) buffer to fetch data from DDR4/HBM memory without interruption (allowing burst transfers) to on-chip memory, and then (2) by caching mesh points using the multiple levels of memory, from registers to MLABs/M20Ks. This is known as implementing a *window buffer* [23]. To implement such a data-reuse path with SYCL, the external/near-chip memory read, computation and write back to external/near-chip memory each was codified as a separate SYCL kernel with `sycl::pipes`, a DPC++ extension, used to move data between the kernels. These kernels operate in parallel. A basic window buffer setup can be seen in Listing 6 lines 11–12 and lines 17–20 where we are reading and writing values such that a certain length of data is buffered in the window.

Unrolling Iterative (time-marching) loop / Step Parallel method

Unrolling the iterative loop as in Chapter 3 will instantiate the same stencil loop's computation many times. In C++ for Vivado, it can be function encapsulating stencil loop can be instantiated many time and data flow optimisation can be applied for parallel operation of stencil loops. when targeting intel FPGAs using SYCL, HLS tool doesn't apply data flow optimisations across multiple loops. In-order to execute two loops in parallel on FPGA, they should be encapsulated in separate kernels. Thus we use a template based stencil computation function to produce unique names (see Listing 8, line 2). While unique kernel names are optional in SYCL 2020, unique kernel names are used

```
using namespace sycl;
template <int id> struct stencil_compute_id;
template<int idx, int DMAX, int VFACTOR>
void stencil_compute(queue &q, int size0, int size1){
 q.submit([&] (handler& h){
 h.single_task<class stencil_compute_id<idx>> ([=] (){
  ... // declarations and setups
  for(int i = 0; i < size0/VFACTOR*(size1+D/2); i++){
   if(cond1) vec_r =pipeS::PipeAt<idx>::read();
   ... // window buffers and stencil computation
   if(cond2) pipeS::PipeAt<idx+1>::write(vec_w);
  }
 });});
}
```

Listing 8: Stencil compute kernel skeleton.

```
template <int N> struct itr_loop {
  static void instantiate(queue &q, int nx, int ny){
    itr_loop<N-1>::instantiate(q, nx, ny);
    stencil_compute<N-1, 4096, 8>(q, nx, ny);
  }
};
template<> struct itr_loop<1>{
  static void instantiate(queue &q, int nx, int ny){
    stencil_compute<0, 4096, 8>(q, nx, ny);
  }
};
```

Listing 9: Pipelining stencil compute kernels.

here to create multiple instances of same kernels on FPGA. As noted before, to move data from one kernel to another, internally, SYCL uses pipes which are similar to streams (e.g. `hls::stream`) in Vivado C++ on Xilinx FPGAs. Thus, a stencil kernel will get input from a pipe and it will push the output to another pipe. As such, pipes should also be unique to indicate the connection between the unique producer/consumer kernels. An indexable pipe array can be created using a struct construct to obtain unique pipes. We use the index from the instantiated template of the stencil compute function for kernels name and choose the pipes as illustrated in Listing 8.

Unrolling of the time-marching loop can be implemented in SYCL using a template based recursive struct function and with a template specialization as in Listing 9. Here we note that `stencil_compute` kernel pops the input data from and pushes the result to the relevant pipes with the adjacent index on the pipe array. Using these techniques we can create a kernel pipeline with any given iterative loop unroll factor of N.

Batching

In Chapter 3, Batching optimisation is applied to amortize the overheads such as compute pipeline latency and kernel call overheads. Same optimisation is required on intel FPGAs

```
[[intel::disable_loop_pipelining]]
for(int itr = 0; itr < n_iter; itr++) {
  accessor ptrR = ((itr & 1) == 0) ? in : out;
  accessor ptrW = ((itr & 1) == 1) ? in : out;
  [[intel::ivdep]] [[intel::initiation_interval(1)]]
  for(int i = 0; i < total_itr+delay; i++) {
    struct dPath16 vecR = ptrR[i+delay];
    if(i < total_itr) pipeM::PipeAt<idx1>::write(vecR);
    struct dPath16 vecW;
    if(i >= delay) vecW = pipeM::PipeAt<idx2>::read();
    ptrW[i] = vecW;
  }
}
```

Listing 10: Global memory read-write loop.

using sycl and No special techniques in SYCL are required to program batching. We discuss and quantify the performance implications of batching later in section 5.2.1

Reducing kernel call overhead

While batching provides a reasonably good way to hide kernel call overheads, it requires large batch sizes to be effective. For example for the RTM application, we benchmark later in this Chapter, a 3D mesh of size $32 \times 32 \times 32$ requires a batch size of 1000 (i.e. 1000 meshes) to hide kernel call latency. A general solution for this problem is to move the time-marching loop to the FPGA(Chapter 3). When the host executes the time-marching loop, the read and write kernels are called by swapping the memory locations and the runtime can schedule the read kernel and write kernel at the same time as each access different data structures. In this case, the host must wait until the completion of the dependant kernels, providing an implicit sync point. However, if the time-marching loop is moved to the device (i.e. FPGA), then both the read and write kernels must be called with both the read and write memory locations together with a signal/flag (through a pipe) to notify the read kernel that the write kernel has completed writing to the specified memory locations. In this case, the SYCL runtime notes this as a data dependency, leading to a deadlock. The run-time waits for the write module to complete first before scheduling the read kernel, hanging the kernel pipeline. In contrast on Xilinx FPGAs using C++ for Vivado in Chapter 3, such a deadlock does not occur as a more hand-tuned complete data-flow path can be created, from read, compute, to write, within the iterative loop in a single kernel.

A solution can be attempted to avoid a deadlock, by fusing global memory read and write accesses into one nested loop as in Listing 10, creating a single kernel. Attribute `intel::ivdep` is used to instruct the compiler that there are no memory access dependencies in the inner loop allowing the compiler to fully pipeline the inner loop. Pipelining is disabled for the outer loop due to data dependency between iterations as the inner loop's read and write locations are swapped in each iteration. However, this implementation will also result in a deadlock or poor performance if pipe `read` is not delayed correctly. To understand the issues we must look at how statements inside a kernel are scheduled and how an iteration of a loop moves through the stages of the compute pipeline for each clock cycle on the FPGA.

In Listing 10, Pipe `read` and `write` are blocking operations, where the loop iteration will not continue until these operations complete. Here `vecR` is loaded from memory and will be pushed to the pipe and go through the compute pipeline before returning as an output result `vecW` through another pipe for `read`. There are a number of clock cycles between the first push of `vecR` and first pop of `vecW`. Assume, for example, `read` (*rd*) and `write` (*wr*) are scheduled at the 10th and 800th clock cycles (exact clock schedules for *rd* and *wr* can be obtained from the kernel schedule viewer section of the report generated by Intel's DPC++ compiler). Then for loop iteration 0, `rd` and *wr* operations are scheduled at the 10th and 800th clocks and for iteration 1 they are scheduled at 11th and 801st clocks and so on. If the first *rd* is not successful until the 50th cycle, then first `wr` can only occur in the 840th cycle. Such blocking can be avoided by introducing a delay as done in the conditional statement on line 10 in Listing 10.

To calculate the required delay, assume *rd, wr* operations are scheduled at clk_{rd} and clk_{wr} and there are S number of pipeline stages between pipe `idx1` and pipe `idx2`. In this case, data pushed to pipe `idx1` will come back to pipe `idx2` only after S clock cycles. Hence, we have to delay the pipe `read` by delay $d >= clk_{rd} - clk_{wr} + S$ number of loop iterations to avoid stalling of pipeline (here we assume uniform data path width across kernels). If delay $d >= clk_{rd} - clk_{wr}$ and $d < clk_{rd} - clk_{wr} + S$ then there will be stalling, but loop will continue as data will be available after a fewer clock cycles than expected, leading to reduced throughput. In case of delay $d < clk_{rd} - clkwr$, then the implementation will deadlock, as data is expected from `read` pipe (`idx2`) before it is pushed to the pipe `idx1`. A further consideration for a 2D stencil computation as in Listing 6, is that at least a row of elements are required to start the computation and return the first output. This adds an additional delay which we note as a buffer delay d_b, leading to a total delay $d >= clk_{wr} - clk_{rd} + S + d_b$ to avoid stalling. Again $d < clk_{wr} - clk_{rd} + d_b$ will result in a deadlock as pushed data will never be available at the time a `read` is attempted and it stalls the whole loop iteration leading to no new data also being pushed to the `write` pipe.

5.2.1 Performance Model

Section 3.2 developed models for 2D and 3D stencil applications to analytically predict the total runtime of the time-marching loop on Xilinx FPGAs. The same terms are used here for the SYCL design on Intel FPGAs with the addition of the schedule delays discussed in the previous section. Schedule delays are usually small when the mesh size is reasonably large but it is significant compared to the processing time for smaller meshes with small batch sizes. The total delay for a 2D stencil application can be modeled as follows:

$$delay_{2D} = (S_{2D} + d_{b,2D}) \tag{5.1}$$

$$S_{2D} = \sum_{i=0}^{kernels} (clk_{wr,i} - clk_{rd,i}) \tag{5.2}$$

$$d_{b,2D} = \left\lceil \frac{m}{V} \right\rceil \times p \times \frac{D}{2} \tag{5.3}$$

Here the mesh size in each dimension is given by m, n and V, p, D are vectorization factor, iterative loop unroll factor and stencil order respectively. $clk_{wr,i}, clk_{rd,i}$ are the clock cycles where pipe `write` and pipe `read` are scheduled in the i^{th} kernel. Here we note that, pipe width is V for all kernels. Similarly for a 3D application the delay can be modeled as in equation (5.4) when the size of the 3rd dimension is l.

$$delay_{3D} = (S_{3D} + d_{b,3D}) \tag{5.4}$$

$$S_{3D} = \sum_{i=0}^{kernels} (clk_{wr,i} - clk_{rd,i}) \tag{5.5}$$

$$d_{b\_3D} = \left\lceil \frac{m}{V} \right\rceil \times n \times p \times \frac{D}{2} \tag{5.6}$$

Adding the above delays to the latency for processing B number of meshes (i.e. batch size) from (Chapter 3) gives the total latency of a 2D and 3D application as equations (5.7) and (5.8) respectively:

$$Clks_{2D} = \frac{n_{iter}}{p} \times \left(\left\lceil \frac{m}{V} \right\rceil \times n \times B + delay_{2D} \right) \tag{5.7}$$

$$Clks_{3D} = \frac{n_{iter}}{p} \times \left(\left\lceil \frac{m}{V} \right\rceil \times n \times l \times B + delay_{3D} \right) \tag{5.8}$$

We make use of the models developed here to predict the performance of the applications benchmarked in Section 5.4.

5.3 Multi-Dimensional Tridiagonal Solvers

Chapter 4 explored workflow for Multi-Dimensional Tridiagonal solvers on Xilinx FPGAs. Various algorithms for implementing multi-dimensional tridiagonal solvers on FPGAs were explored there, focusing on gaining higher throughput from multiple solvers setup on a multi-dimensional domain. Analytical models presented in Section 4.1 reveals that inexpensive Thomas solver(Algo 1) is most efficient algorithm when there is multiple systems to be solved even-though it suffers from intra-loop and inter loop dependencies. This section explores Thomas solver algorithm on intel FPGAs together with the utility of SYCL for their synthesis on Intel FPGAs.

The Thomas algorithm carries out a specialized form of Gaussian elimination (assuming non-zero b_i) providing the least computationally expensive solution, but suffers from a loop carried dependency. It has a time complexity of $O(N)$. Implementing Alg. 1 using floating-point primitive cores on an Intel FPGA such as the PAC D5005 would incur an arithmetic pipeline latency of 37 clock cycles for the forward path and 6 clock cycles for the backward path. The forward path cycles are dominated by the slow floating-point division operations (26 clks latency for division operation). This essentially gives a dependency distance (l_f) of about 37 cycles (taking the maximum out of the forward loop and backward loops latencies). Then, Chapter 4 demonstrates how $g = l_f$ number of tridiagonal systems can be grouped and interleaved such that iteration 1 of system 1 is input to the pipeline, followed by iteration 1 of system 2 and so on, per clock cycle, up to iteration 1 of system g. This allows to obtain higher throughput by continuously utilizing the computational pipeline versus solving one system at a time. Techniques such as double buffering (i.e. ping-pong buffers), can be used to further optimize the implementation. With SYCL, this can be codified with three kernels, one each for forward and backward loops and one for the interleaving of the systems.

With a minimum group size of $g_f = 37$, on the Intel PAC D5005, to solve systems with size N, with interleaving, the Thomas solver would require four on-chip block RAMs (for a, b, c, d) with $2 \times g_f \times N$ number of words, totaling $8 \times g_f \times N$ words. The $2\times$ is due to the need of twice as much memory to setup ping-pong buffers. Storage for c^*, d^* for the forward pass kernel will require two RAMs with $2 \times g_f \times N$, totaling $4 \times g_f \times N$ words. Additionally storage for u in backward pass would require a RAM with $2 \times g_b \times N$ words. As such a total of $456 \times N$ words is required for the Thomas solver implementation on this specific FPGA. Additional RAMs with a smaller number of words are required for storing the previous iteration values as detailed in Chapter 4.

In some applications, the coefficients a, b, c can be generated without reading from external memory, using an initialization routine. In these cases, coefficients a, b, c need not be interleaved, but instead could be calculated as part of the interleaving kernel. Fusing this coefficient generation reduces total memory cost to $234 \times N$ words. A further saving of on-chip memory can be done for Intel FPGAs by separating the calculation of r into a separate kernel which we denote as a `r_generator` kernel. This can be done for Alg. 1 by creating a kernel with only lines 4 and 6. Line 5 will be a separate kernel that gets the computed value of r through a pipe. Calculating r only requires coefficients a, b, c which again can be internally calculated within the `r_generator` kernel. Now, the `r_generator` kernel would require group size $g_r = 37$, `Thomas_forward` and `interleave` kernels would require group size of $g_f = 9$. The group size of `Thomas_backward` remains same. This optimisation reduces the total memory cost to $140 \times N$ words. It is a 69% reduction from the non-fused version and a 40% reduction compared to a fused variant with no `r_generator` in Chapter 4.

The Thomas solver can be vectorized to solve multiple systems in parallel. Again this can be done using a wider data path using arrays inside *struct* as illustrated in Section 5.2's `struct dPath16`. In our implementation, template parameters are used to specify the vectorization factor, data type, group size and input and output pipe index of the pipe

Table 5.1: Experimental systems specifications.

FPGA	Intel PAC D5005 [29]
DSP blocks	5760
MLABs / M20K	7.6MB / 29.3 MB
DDR4	64GB, 76.8GB/s, in 4 banks (1 channel/bank)
Host	Intel Xeon Platinum 8256 @3.8GHz (16 CPUs, 4 cores each) 1559 GB RAM, Ubuntu 18.04.6 LTS
Design SW	Intel oneAPI 2021.4.0, Intel Quartus software 19.2
board_variant	pac_s10
GPU	**Nvidia Tesla V100 PCIe [53]**
Global Mem.	16GB HBM2, 900GB/s
Host	Intel Xeon Gold 6252 @2.10GHz (48 cores) 256GB RAM, Ubuntu 18.04.3 LTS
Compilers, OS	nvcc CUDA 10.0.130, Debian 9.11

array. We use a manually flattened loop with custom ping-pong buffers to save device resources using custom integer data types for loop control and improve the latency by continuous execution than a nested loop based ping-pong buffer implementation. However, manually flattened loops require a dependency distance that can be specified using the `[[intel::ivdep(safelen)]]` attribute.

5.4 Performance

This section presents the experimental results of applying our design strategy by implementing two non-trivial, representative applications using SYCL on Intel FPGAs. The first application is a stencil application implementing an explicit numerical solver and the second application uses multi-dimensional tridiagonal solvers. The applications are synthesized on an Intel PAC D5005. Non-Unified Shared Memory (USM) model of SYCL is used as it provides a simpler memory access implementation compared to the alternative USM model [65]. Finally, the performance of the applications on the FPGA is compared to equivalent implementations of the same applications, written in CUDA [77], on an Nvidia V100 GPU (raw runtime values are available in Appendix section C.3). Specification of FPGA and GPU Systems along with the specific software tools used in the evaluation are detailed in Table 5.1.

5.4.1 Reverse Time Migration (RTM) Forward-Pass

In the section 3.4, we compared the performance of 3D RTM application(Algorithm 3) on Xilinx U280 FPGA and Nvidia V100 GPU. This section compares the same application implementation on Intel PAC 5005 FPGA and Nvidia V100. This 3D application using higher order stencils on vectored elements requires access to multiple data structures.

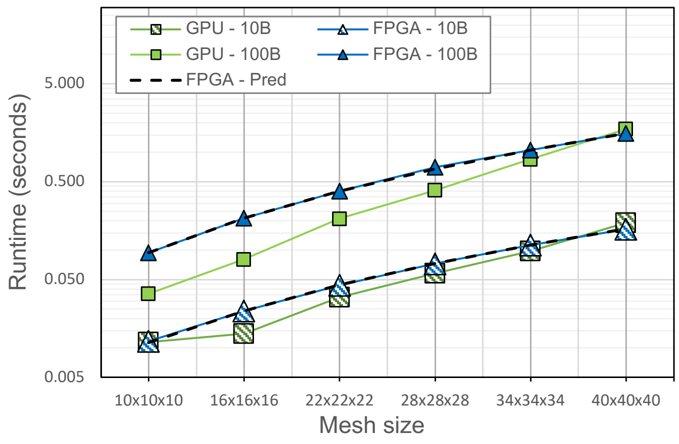


Figure 5.1: RTM forward-pass, FP32, $p = 2, v = 3$, 200 iterations.

Fused loops along with kernel-to-kernel data movement using SYCL pipes reduce the required off-chip memory bandwidth within the available 76.8 GB/s limit. Available DSP resources with native support for single precision operations better utilised by unrolling the iterative loop.

Managing the on-chip memory to enable the execution of larger mesh sizes is a challenge for higher order stencil applications with vector mesh elements. We attempted to maximize the vectorization factor to reduce the iterative unroll factor to save on-chip memory while maintaining the same compute throughput. The maximum possible vectorization factor is 4, due to the off-chip memory bandwidth limitation. We opted to set this to 3, as it then allowed us to have an iterative loop unroll factor of 2 as well (a vectorization factor of 4 and an unroll factor of 2 would result in a design that hits the upper limit of the available DSP units). In Chapter 3, RTM application is implemented on Xilinx's U280 FPGA but vectorization isn't attempted. This was due to the organization of the U280 into Super Logic Regions (SLRs), with each single SLR not having sufficient DSP resources for such an implementation.

Application runtimes are detailed in Figure 5.1. Mesh sizes from 10^3 to 40^3 are executed with batch sizes (B) of 10 and 100. The model is also used to predict the runtime (dotted line noted as FPGA-Pred) for each case and the prediction error is below 5%. It shows that the FPGA performance is on par or better compared to the V100 GPU performance. We attribute this to the availability of a larger number of DSP units with native support for floating-point operations. Table 5.2 compares the effective bandwidth and energy consumption on the FPGA with the GPU. Bandwidth Utilisation is provided for GPU which mainly depends on global memory. FPGA's bandwidth utilisation is underpinned by the fast on-chip memory performance (with tens of TB/s) which we do not show here. As such, the FPGA's effective bandwidth reaches up to 418GB/s even though global memory bandwidth is limited to 76.8 GB/s. Utilizing fast on-chip memory performance is a direct consequence of using window buffers and communication between stencil compute kernels via pipes. GPU reaches bandwidths of up to 681GB/s for the

Table 5.2: RTM - Avg. Bandwidth, BW (GB/s), Avg. Utilisation(%) and Energy, E (J), 200 iters.

Mesh	BW-10B		BW-100B		E-100B	
	FPGA	GPU	FPGA	GPU	FPGA	GPU
10^3	154	158 (18%)	192	506 (56%)	8.5	4.8
16^3	230	391 (43%)	258	681 (76%)	19.4	12.9
22^3	286	379 (42%)	313	598 (66%)	36.2	37.7
28^3	331	414 (46%)	342	588 (65%)	62.9	75.2
34^3	367	420 (47%)	390	486 (54%)	96.2	164.6
40^3	397	344 (38%)	418	379 (52%)	141.3	344.7

16^3 utilizing higher portions of its peak bandwidth. This indicates near-optimal performance from the GPU implementation. We explored both Array of Structure (AoS) and Structure of Arrays (SoA) data layout for the vector elements on the GPU. SoA gives the best throughput due to better data access patterns. We speculate that the reduced performance for larger mesh sizes is due to poor cache utilization when solving higher order stencils.

We used the `fpgainfo` utility to measure power consumption on the PAC D5005. The utility gives voltages and current of the 12V PCIe power supply as well as 12V AUX power supply. GPU power consumption is obtained through `nvidia-smi`. For RTM, the power consumption of the Intel PAC D5005 is between 84-94W while the V100 GPU's power consumption is between 47–200W. Observed power-draw indicate that the FPGA is just over 59% less energy consuming than the GPU for the largest mesh with the larger batch sizes.

5.4.2 ADI 2D Heat Diffusion Application

Algorithm 6: 2D ADI Heat Diffusion Application

```
1: for $i = 0, i < n_{iter}, i++$ do
2:     Calculate RHS : $d = f_{7pt}(u), a = \frac{-1}{2}\gamma, b = \gamma, c = \frac{-1}{2}\gamma$
3:     Tridslv(x-dim), update $d$
4:     Tridslv(y-dim), update $d$
5:     $u = u + d$
6: end for
```

As in Chapter 3, Alternating Direction Implicit (ADI) time discretization requires multiple tridiagonal systems to be solved in multiple dimensions. This section compare implementation of the 2D heat diffusion equation using ADI on FPGA and GPU. The High-Level algorithm is detailed in Alg. 6. Performance on FPGAs could be maximized by pipelining all four steps in Alg. 6 due limitations in global memory bandwidth. Intermediate results from *Tridslv(x-dim)* will need to be transposed using on-chip memory to achieve this. Once all the kernels are pipelined, the iterative loop can also be unrolled. The implementation in Chapter 4, on the Xilinx U280 for the same application, used an unroll factor of 3 and then scaled the design to multiple compute units (CUs) based on available HBM ports. Scaling to multiple CUs, instead of using a higher unroll factor results in lower latency for small batch sizes. Since HBM memory is not available on the Intel D5005, we preferred to unroll the iterative loop instead of scaling to CUs in the present work. The performance model for this implementation including the scheduling latency (due to the existence of a stencil loop) can be noted as in equation (5.9):

$$L_{adi,2D} = (n_{iter}/f_U) \times L_{rhs+xy} \tag{5.9}$$

$$\begin{aligned} L_{rhs+xy} =& f_U \times [(2x/V) + (2vx/V + 3gx) + (2xy/V + 3gy)] + \\ & B \times (xy/V) + \sum_{i=0}^{kernels} (clk_{wr,i} - clk_{rd,i}) \end{aligned} \tag{5.10}$$

Here, x, y are mesh sizes, B is the batch size, V is the vectorization factor, g is the group size of systems, f_U is the unroll factor of the iterative loop. This is similar to the models created for the 2D ADI application in Chapter 4.

Figure 5.2 gives runtime performance of 2D ADI Heat Diffusion application in FP32 on the Intel PAC D5005 and compares it to performance on the Nvidia V100 GPU. Even though the FPGA implementation operates at 231MHz, it outperforms the GPU. We attribute this to the unrolling of the iterative loop by a factor of 8 and the fusion of coefficient in the Thomas solver. This essentially allows data to be kept on faster on-chip memory without writing to global (external) memory for the whole computation. The same type of fusion is not supported by the GPU implementation as the GPU tridiagonal solver [77] call is a function call to an external library. Additionally, it does not support fusion of coefficient generation internally.

We can estimate the runtime of the GPU if coefficients were generated internally, assuming that the GPU is not compute limited and the same sustained bandwidth is maintained for each of the application cases. Estimated Run times for each kernel call can be computed using equation (5.11). Here, t_{opt}, t are run-times for the GPU implementations with and without internally generated coefficients respectively. When generating coefficients internally, data movement does not include data structures coefficient meshes a, b, c. Then the run-time estimate for the full ADI application is can be obtained from equation (5.12), where runtimes are adjusted for the RHS calculation, *Tridslv(x-dim)* and *Tridslv(y-dim)*. Here, we note that the accumulation step in Alg. 6 is fused into *Tridslv(y-dim)* of the GPU implementation.

$$t_{opt} = t \times data\_movement_{opt}/data\_movement \tag{5.11}$$

$$t\_adi_{opt} = \frac{1}{4} \times t_{preproc} + \frac{2}{5} \times t_{xsolve} + \frac{4}{7} \times t_{ysolve} \tag{5.12}$$

Even when the coefficients are internally generated on the GPU, the FPGA appears to perform marginally better, as can be seen by the dotted red lines in Figure 5.2. The model predicted runtimes for FPGA is closely matching with the actual runtimes with a

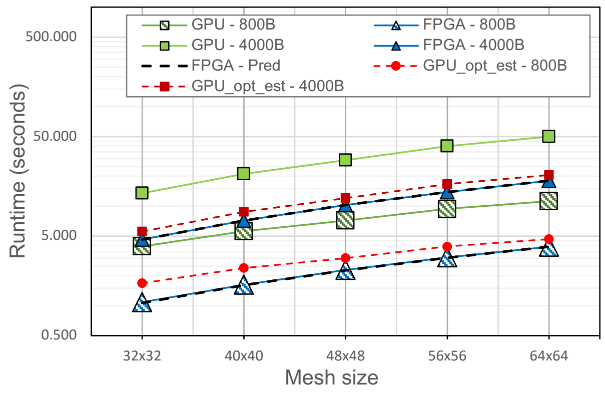


Figure 5.2: ADI 2D, FP32, $f_U = 8, V = 8$, 16000 iter meshes.

Table 5.3: ADI Heat Diffusion Application: Achieved Bandwidth, BW (GB/s) Utilisation(%) and Energy, E (KJ)

2D FP32 (16000 iterations, $f_U = 8$), F - FPGA, G - GPU								
Mesh	BW-800B			BW-4000B			E-4000B	
	F	Gx	Gy	F	Gx	Gy	F	G
32^2	386	131 (15%)	288 (32%)	453	185 (21%)	524 (58%)	0.453	1.825
40^2	403	144 (16%)	330 (37%)	457	197 (22%)	478 (53%)	0.712	3.159
48^2	414	163 (18%)	389 (43%)	460	202 (22%)	517 (57%)	1.032	4.363
56^2	422	169 (19%)	412 (46%)	462	202 (22%)	498 (55%)	1.411	6.220
64^2	428	182 (20%)	477 (53%)	463	205 (23%)	555 (62%)	1.820	7.580

prediction error of less than 2%.

Table 5.3 details the effective bandwidth of the FPGA and GPUs with bandwidth utilisation for GPU and Energy consumption for both devices. The GPU bandwidth is noted for x and y solves separately given that these are separate calls to the tridiagonal solver library. The FPGA bandwidth reaches up to 463GB/s. This, as noted above, is due to on-chip memory based data movement without reading/writing from lower bandwidth global memory. In GPU, *Tridslv(y-dim)* reaches a good bandwidth of 555 GB/s but *Tridslv(x-dim)* performs poorly, only reaching up to 205 GB/s. Such lower bandwidths are also reported by [42] due to the 8×8 transpose operations using registers/shared memory on GPUs. FPGA power consumption varies between $95 - 101$W while the GPU power consumption varies between $105 - 151$W. for the largest mesh with running the largest batch size, the FPGA saves over 76% energy used compared to the GPU. The same application on the U280 used HBM based delay buffers [15] to save on-chip memory and managed to run mesh sizes of up to 128×128. On the Intel PAC D5005, larger delay buffers are also implemented using on-chip memory and this limits the largest mesh size to 64×64.

5.5 Concluding Remarks

In this Chapter, we explored the design and development of structured-mesh based solvers using SYCL for Intel FPGA hardware. Two classes of applications were targeted (1) stencil applications based on explicit iterative methods and (2) multi-dimensional tridiagonal solvers based on implicit methods. A generalized workflow, extending the work in Chapter 3-4, for synthesizing optimized solvers of these applications was developed together with an analytic model to predict their performance in support of design space explorations. The extensions targeted key optimizations required to obtain the best performance using SYCL programming techniques. The main methods for improving performance with SYCL included (1) reducing SYCL kernel calling overhead by moving the time-marching outer loop onto the FPGA device and (2) reducing on-chip memory usage for the Thomas solver for implementing multi-dimensional tridiagonal solvers. The designs and workflow were applied to two non-trivial applications synthesizing them on an Intel PAC D5005 FPGA. Performance results were compared to the same applications implemented on an Nvidia V100 GPU as a baseline. Observed results indicate the FPGA provided better or matching performance compared to the V100 GPU in terms of runtime. We also see 59%–76% less power consumption when executing these applications at their largest mesh and batch sizes. The performance models provided high accuracy with less than 5% model prediction errors for all cases. Future work will extend these techniques to other Intel FPGAs with HBM memory and also consider applications with larger meshes that were currently limited by the PAC D5005's hardware resources. We will also compare the performance of our multi-dimensional tridiagonal solver design to Intel's tridiagonal solver library.

The significant effort in applying non-trivial transformation to optimize the SYCL implementations demonstrates the programming overheads still dominating development on FPGAs. This is still true even with the hardware vendors providing mature HLS tools for development. While we have not quantified the productivity overheads in this thesis, it is clear that such hand-tuned, hardware specific programming is not tractable particularly for developing and maintaining codes for execution on multiple hardware platforms such as GPUs and FPGAs, even with language extensions such as SYCL.

Chapter 6

Towards Automating FPGA Designs

High-level workflow to target structured mesh based explicit and implicit numerical applications on FPGA devices have been detailed in previous Chapters. Profitability of FPGAs in terms of time to solution (latency), throughput and energy consumption has been demonstrated. These benefits come at the cost of longer development time and expertise in hardware architecture design/dataflow programming model to implement these applications on FPGAs. Additionally, finding optimal design parameters requires tedious analysis considering FPGA's specification, kernel's stencil computation, as well as overall data movement. Several FPGA specific transformations explained in previous Chapters, such as full data reuse, batching and tiling are not trivial to implement with an FPGA target language such as C/C++ or SYCL. The unified workflow presented in this thesis eliminates the time to re-evaluate these optimisations by domain scientists. However, when targeting FPGAs from multiple vendors, different `PRAGMAS` or `Attributes` needs to be applied. Additionally, a novice FPGA developer (e.g. domain scientist) would have to spend a significant amount of time and effort learning these optimisations and implementing the applications using FPGA target languages. Learning specific FPGA transformations as well as vendor-specific configurations and instructions would have a steep learning curve. An automatic translator/code-generator could significantly reduce the time to implement this workflow, ideally eliminating manual tuning.

Previous works have attempted to automate the design and development of applications for parallel computing architectures. A recent development more widely used for performance portability on traditional architectures such as multi-core/many-core processors is a separation of concerns approach – separating problem specification from its implementations [56, 45, 30, 50, 61, 92, 26]. A domain scientist specifies the application using a DSL, and automatic techniques such as code-generation/translation are used to obtain highly optimized target implementations. A user can explore the performance of the application on different accelerator platforms using code generation and can choose the best accelerator for their requirement. Further, the separation of concerns approach allows the same DSL framework to be used on future devices, just the translator needs to be updated in compliance with specifications and required optimization of future accelerator devices.

Several previous works have attempted to automatically generate FPGA target implementations for structured mesh based applications using domain specific frameworks. SODA [10] provides a DSL by which multiple stencil loops can be chained together and be implemented on an FPGA. It also provides the variables to set the design parameters such as the vectorization factor and the iterative loop unroll factor. SODA is based on a declarative programming model and it is not clear that complex stencil applications can be easily ported to SODA DSL. Moreover, FPGA implementations with SODA require a specific data layout with padding and the host has to reorganise the mesh-points for each iteration for the FPGA kernel to obtain correct data points.

HeteroCL [40] is another domain specific framework for stencil solvers and other popular domains of applications (Image processing, Neural networks) with a python based DSL. HeteroCL attempts to decouple the algorithm from the hardware customisation such as the ones arising in computations (e.g coalescing the loops, loop fusion), data type and memory architecture. In this way, it facilitates design space exploration while keeping the application specification unchanged. In contrast to SODA, HetroCL supports both declarative and imperative programming and supports a wider domain of applications. Unlike the DSL for classical accelerators, HeteroCL supports bit-accurate data types to obtain better Quality of Results (QoR) on FPGAs. HertroCL offers a general backend and utilizes two specialized backends, SODA [10] for stencil computations and PolySA [12] for systolic array-based computations. Users can choose the code generation through the specialized backends using a macro or else FPGA target implementation will be generated using the general backend.

The more recent, StencilFlow [15] targets graph like data dependent stencil kernels on multi FPGAs. It extends the Stateful DataFlow multiGraph (SDFG) intermediate representation of DaCe [6] framework and generates HLS target code using the DaCe framework. Like HeteroCL, DaCe targets to separate program definition from optimisation. Stencil application written using StencilFlow will be lowered to SDFG intermediate representation and performance optimisation can be applied to it. In this way, DaCe targets to separate program specification from performance optimisation. Optimisations include fusing loops, expanding stencil computation nodes for data reuse and adding delay buffers nodes to avoid deadlock in circular branches.

In spite of the above works, users face several obstacles when attempting to employ FPGAs. Firstly, optimizations such as batching, tiling and integration of tridiagonal solvers, particularly for real-world, non-trivial applications are not immediately supported by the current frameworks. Secondly, porting complex applications using declarative programming would require significant time and effort. Instead, users would prefer more familiar imperative languages (C++/Python) based embedded DSLs. Finally, a domain scientist may prefer a framework which supports automatic code generation for many accelerator platforms including FPGAs for a more straightforward exploration of performance on multiple platforms using the same DSL.

To this end, in this final Chapter, we use the OPS (Oxford Parallel Library for Structured-mesh solvers) DSL for structured mesh-based numerical applications for proposing a path to automation. OPS has the following features that support our development:

- The DSL separates computation, communication, and data structures, enabling effective optimization for each of these aspects independently on FPGAs.
- By providing APIs/constructs for stencil computation and multi-dimensional tridiagonal system solving, along with explicit annotations on inputs, outputs, stencil specifications, data access modes, and mesh dimensions, it limits implementation to a small set of APIs/constructs while supporting a broader range of structured mesh-based applications.
- OPS already supports a variety of accelerator platforms, including GPUs and CPUs, which will minimize the amount of time users spend learning about a new framework to target FPGAs. Additionally, it allows for the exploration of benefits and comparison of implementations using the same DSL-based application.
- OPS code generation is extensible, supporting the integration of new target architectures through a simple and well-organized translator stack.

To bridge the research gap in achieving the near-optimal FPGA implementation through high-level application specification, we employ OPS in this Chapter and establish transformation procedures to execute OPS applications on FPGAs. Although transformation techniques detailed in this Chapter targets C++ for Vivado, the techniques can be applied in a similar manner to obtain SYCL implementations. These steps are designed such that they will enable the creation of an automatic translator using modern compiler frameworks. **The implementation of the compiler stack is not attempted in this work but is left as future work.**

The organization of this Chapter is as follows: The first section provides an overview of OPS, while the subsequent sections outline the transformation methodology for the baseline design described in Chapter 3, followed by the application of optimizations. The later sections present the steps required to implement full OPS applications on FPGAs and detail the procedure for identifying the optimal design parameters automatically.

6.1 OPS Framework for Structured Mesh Applications

OPS is a C/C++ based embedded DSL to target the domain of structured mesh-based numerical applications. Applications in this domain are characterised by looping over rectangular meshes where connectivity is implicit. OPS utilizes the characteristics of this application class and decomposes the applications into abstract parts such as mesh data,

```c
int halo_neg[] = {-1,-1}; //negative block halo
int halo_pos[] = {1,1}; //positive block halo
int size[] = {10,20};
int base[] = {0,0};
double* d1 = malloc(...)
double* d2 = malloc(...)
ops_block A = ops_decl_block(2, "A");

ops_dat dat1 = ops_decl_dat(A,1, size, base, \
    halo_pos, halo_neg, d1, "double", "dat1");

ops_dat dat2 = ops_decl_dat(A,1, size, base, \
    halo_pos, halo_neg, d1, "double", "dat2");
```

Listing 11: OPS block and datasets.

stencils defining how data is accessed and computation over the accessed data. While OPS supports multi-blocks to facilitate computation over complex shapes, the focus of this thesis is limited to single blocks.

Implementation of the applications is supported through high-level domain-specific APIs, which will appear as a classical software library for the developers. OPS then uses a source-to-source translator to parse the APIs and generate parallel implementations. This generated implementation is linked against specific parallel library backend implementations for execution on accelerator devices.

6.1.1 OPS API

A typical OPS application is developed as a sequential program. This makes numerical validation and debugging more straightforward, while high-level specifications make the resulting implementation easy to comprehend and maintain. As OPS is designed for multi-block problems, it requires datasets to be assigned to certain blocks. An OPS block is defined with dimensionality and a name string. Multiple datasets can be attached to the OPS block, as shown in Listing 11. The function `ops_decl_dat(...)` attaches the datasets (`d1, d2`) to the block (`A`), and takes additional parameters such as the number of values for each point (1 in this case), the dimensions (`size`) of the mesh, the starting point of actual data (`base`), the sizes of the block halos (`halo_neg, halo_pos`), the data type of values on each mesh point (`double`), and name strings (`"dat1", "dat2"`) for the data.

Another way of defining an `ops_dat` is through `ops_decl_dat_hdf5`, which reads the data directly from an HDF5 file. A null pointer can also be passed to `ops_decl_dat(...)` if initialized data is not required. OPS takes ownership of data fields declared through `ops_decl_dat(...)`, and can reorganize the data structure to optimize acceleration on the target platform.

The key assumption that makes the separation and re-organization of data possible is that the result of the computation may not depend on the order in which primitive computations are carried out on mesh points. This gives OPS the freedom to execute the computation using various optimization and parallelization techniques. However, there are a few exceptional cases, specifically when solving implicit applications using tridiagonal system solvers, which are order-dependent algorithms.

A typical structured mesh-based stencil application can be described as an operation over the mesh points on the data structures in the given block. This corresponds to a reading set of mesh points specified by stencils, doing computations on the read data, and writing back results through a stencil. Such a simple application is shown in Listing 12.

```
int Range[4] = {1, 9, 1, 19};

for(int i = Range[2];  i < Range[3]; i++){
    for(int j = Range[0]; j < Range[1]; j++){
        d2[i][j] = d1[i][j] + d1[i][j+1] + d1[i+1][j] + d1[i][j-1] + \
                   d1[i-1][j];
    }
}
```

Listing 12: A Stencil computation loop.

The loop in Listing 12 can be mapped to the OPS application as in Listing 13 with the separation of data structures and computation which is specified through a kernel function. The function `calc` is called the user kernel in OPS terminology, as it is specified by the domain scientist to apply over data points. Here, we can clearly see that the computation is specified without any indication of how it will be paralleized on a target architecture. Additionally, `ops_par_loop` specifies the block the computation is to be attached to, its dimension, ranges over each dimension, and the `ops_dats` involved in the computation. `OPS_ACC0` and `OPS_ACC1` are access macros that convert to corresponding indexes to access a contiguous memory block allocated by OPS internally for each `ops_dat`. Although the specified stencils are not directly used in such index calculations, they are useful for error-checking memory access. Moreover, `OPS_WRITE` and `OPS_READ` convey information on access modes on each `ops_dat`, which could be used to efficiently implement memory access with different optimizations. Similar to `ops_arg_dat`, `ops_arg_gbl()` can be used when there is a need for a global reduction on a certain variable. Further details of the OPS Application Programming Interfaces (APIs) can be found in the OPS manual [57].

6.1.2 Application Development Using OPS

Input data and corresponding computation are specified through the OPS API as in Listing 13. OPS have the flexibility to rearrange the data set and to do the computation in the desired order so as to get the best performance on the target architecture. OPS employs two key techniques to implement the target application optimally on accelerator devices. The first one is to factor out the common patterns to the backend library, which includes data movements between device and host, solving tridiagonal systems in implicit applications and parallel file IO operations.

```c
/* Stencil declarations */
int st0[] = {0,0};
ops_stencil S0 = ops_decl_stencil(2,1,st0,"00");
int st1[] = {0,0, 0,1, 1,0,-1,0, 0,-1};
ops_stencil S1 = ops_decl_stencil(2,5,st1,"5P");

/* User kernel */
void calc(double *a, const double *b) {
b[OPS_ACC1(0,0)] = a[OPS_ACC0(0,0)] +  a[OPS_ACC0(0,1)]
    + a[OPS_ACC0(1,0)] + a[OPS_ACC0(0,-1)] + a[OPS_ACC0(-1,0)];
}


/* OPS parallel loop */
int range[4] = {1,9,1,19};
ops_par_loop(calc, A, 2, range,
    ops_arg_dat(dat2,S0,"double",OPS_WRITE),
    ops_arg_dat(dat1,S1,"double",OPS_READ));
```

Listing 13: Example OPS application.

The second technique is code generation, through the source-to-source translation technique. it is specifically used to produce the parallel implementation of `ops_par_loop` as implementation is specific to the target architecture and application case. A CPU implementation requires multi-threaded/vectorized implementation and GPU implementation is thread-based and generated code will also include the calls to data movement between the device and host.

Figure 6.1 illustrates the workflow for developing the structured mesh-based application using OPS. As mentioned before, domain scientists develop structured mesh-based applications using OPS APIs. The numerical accuracy of the implementation can be tested on a single core CPU by directly compiling and executing it. OPS provides the header file `ops_seq.h` for this purpose. If it is a correct numerical solution, then the domain scientist can move to target parallel architectures using code translation. During the code translation, the OPS translator parses the APIs and generates parallel implementation along with library calls to backend implementations. The generated code can be compiled using conventional compilers such as `nvcc, icpc, gcc` etc and linked against backend libraries as in Figure 6.1.

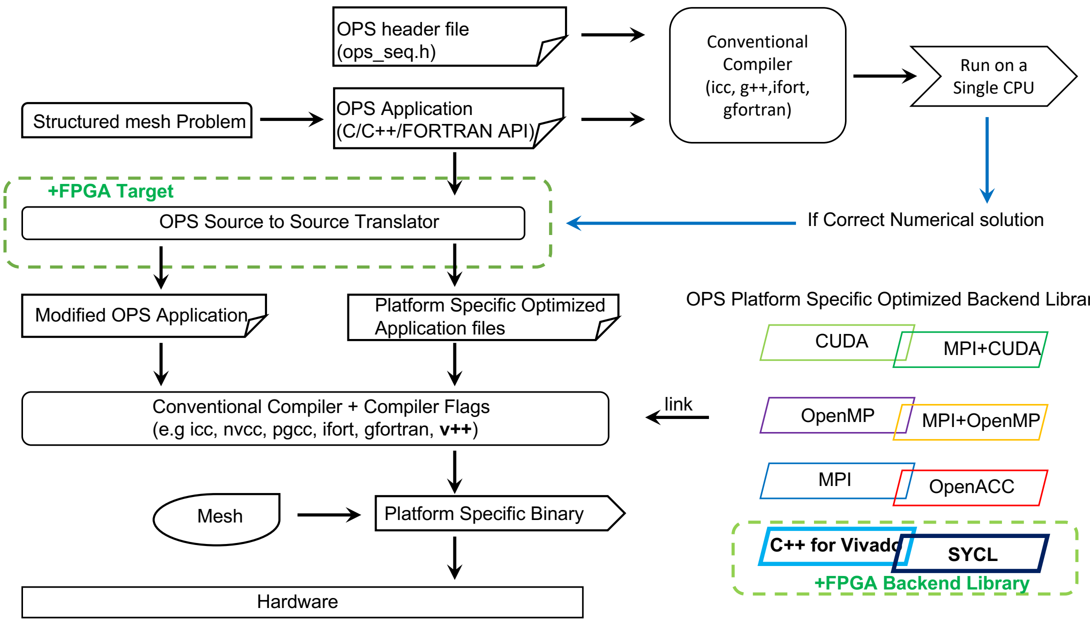

OPS header file (ops_seq.h)
Structured mesh Problem
OPS Application (C/C++/FORTRAN API)
Conventional Compiler (icc, g++,ifort, gfortran)
Run on a Single CPU
+FPGA Target
OPS Source to Source Translator
If Correct Numerical solution
Modified OPS Application
Platform Specific Optimized Application files
OPS Platform Specific Optimized Backend Libraries
CUDA
MPI+CUDA
OpenMP
MPI+OpenMP
MPI
OpenACC
C++ for Vivado
SYCL
+FPGA Backend Library
link
Conventional Compiler + Compiler Flags (e.g icc, nvcc, pgcc, ifort, gfortran, v++)
Mesh
Platform Specific Binary
Hardware

Figure 6.1: The workflow for developing an application with OPS (based on [51]) and how the proposed new FPGA back-end will fit within the framework.

6.2 OPS to FPGA Target transformation

Code generation for FPGA differs significantly from traditional devices (CPUs and GPUs) due to the wider configuration and design space of FPGA kernels and the constraints associated with the fixed amount of resources. As discussed in previous Chapters, unlike kernels for CPUs and GPUs, FPGA kernels implement a tailored architecture (a circuit) for the algorithm, essentially allowing explicit control over internal data movement. However, implementing such architecture requires a certain amount of FPGA resources and must fit within the target FPGA. Loading a kernel into the dynamic region of FPGA takes a few seconds, so it is best to load a set of kernels and use them repeatedly. Unlike CPUs and FPGAs, the performance of kernels on FPGA depends on internal data movement, as off-chip memory bandwidth is limited, as shown in Figure 6.2. Therefore, the performance of a kernel depends on the kernels that feed data to it and take data from it. Therefore, optimization should be applied to a set of kernels as a whole, rather than individually, as is the case with classical accelerators. In order to do collective optimizations on a set of

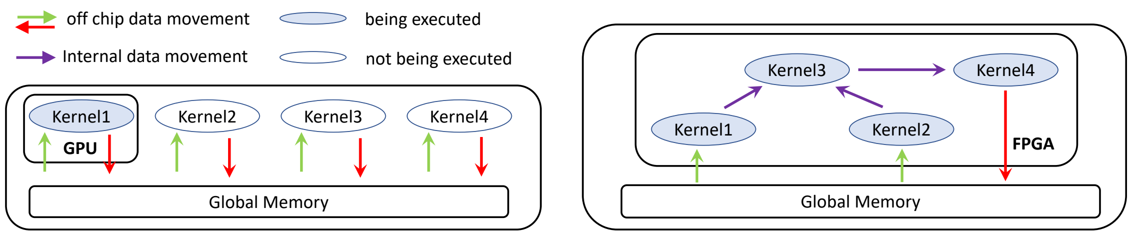


Figure 6.2: kernel execution overview, classical accelerators Vs FPGA

kernels, parallel execution of OPS kernels can be modelled as a dataflow graph based on data dependencies. In such dataflow graphs, nodes process the data and edges represent the data movement. Since the parts of the program accelerated in an OPS application are only specified through `ops_par_loop` for stencil kernels and `ops_tridMultiDimBatch` for solving tridiagonal systems, these two OPS API calls will be mapped to nodes in the generated dataflow graph. Following data structure can be used to represent a node to facilitate the automatic dataflow graph generation.

```
struct Node{int OrderId, List* Inputs, List* Outputs,
    int accDim, List* Indeps,  List* Outdeps};
```

`ops_par_loop` and `ops_tridMultiDimBatch` APIs carries information on input and output `ops_dats`. An `OrderId` value is set for the node based on the order API calls are made. We use the `OrderId` as the primary key for identifying the node. `accDim` is data access dimension, it is always zero for `ops_par_loop` and it corresponds to the dimension along which systems are solved for `ops_tridMultiDimBatch`. `Indeps` and `Outdeps` will be used for matching nodes that provide data and consume data for corresponding inputs and outputs for a particular node. Steps to identifying the list of `Indeps` can be detailed as follows:

1. Take a node (assume `node_x`) from the List of nodes. Loop through each input (assume `dat_x`) of that node and find a dependency node that satisfies the following condition
    - It's `OrderId` is closest but lower to `OrderId` of `node_x`
    - It writes to `dat_x`
    - if a node satisfies the above two conditions, add the node to `deps` list of `node_x`. if no node satisfies the above condition, the corresponding dependency for `dat_x` will be `NULL`.
2. Repeat step 1 until all nodes are looped through

Algorithm 7: 2D ADI Heat Diffusion Application

```
1: for $i = 0, i < n_{iter}, i++$ do
2:     Calculate RHS : $d = f_{7pt}(u), a = \frac{-1}{2}\gamma, b = \gamma, c = \frac{-1}{2}\gamma$
3:     Tridslv(x-dim), update $d$
4:     Tridslv(y-dim), update $d$
5:     $u = u + d$
6: end for
```

In a similar way `Outdeps` for a node can be identified. `Indeps` and `Outdeps` provide the connectivity/dependency between the nodes and a dataflow graph can be generated. As FPGA kernels move the data through internal streams/FIFOs, additional nodes/kernels might be required for the proper execution of the dataflow graph:

- If a node's input dependency is `NULL`, then it should get data from global memory. As we separate computation and communication, a new node will be added to read from global memory. Similarly, if a node's output dependency is `NULL`, a global memory write node will be added. These source and sink nodes in the dataflow graph and detailed in section 6.2.5
- If the access dimension (`accDim`) of the producer node and consumer node are different, then an explicit data reorganization is required through a node between them. We restrict the supported `accDim` to {0,1,2}. If the absolute difference between `accDim` of two nodes is zero, then no additional node between them is required. if the difference is one, a plane transpose is required and it is achieved through a templated library function (refer section 6.2.7). if the difference is two, it would require buffering the entire mesh and is not feasible to do using on chip memory of FPGAs.

Moreover, circular datapath (when there are at least two paths from one node to another) could be found in such dataflow graphs as detailed in section 6.2.6. A stall/deadlock-free execution of such a dataflow graph would require buffers to be inserted in one branch of the circular datapath. In this work, we calls these nodes as delay buffers. Including the delay buffers, there are five key types of node when mapping a OPS application to dataflow graph based computation:

1. `ops_par_loop` nodes
2. global memory access nodes
3. delay buffer nodes
4. `ops_tridMultiDimBatch` nodes
5. data re-organiser nodes

Once the dataflow graph is generated, corresponding nodes need to be implemented using FPGA target language. This thesis proposes the use of implementation skeletons (or templates as used in [4]) for generating the FPGA implementation of nodes in the dataflow graph. The skeletons encode the common program structure, or circuit structure in this case, for each node, which can then be reused for instantiating the concrete implementations of specific nodes in a given program. The use of the skeletons in the overall workflow is illustrated in Figure 6.3. In the following sections, the implementation skeletons for the above different types of nodes is illustrated, starting from `ops_par_loop` nodes. Once each of them are implemented, can be made to execute in parallel by applying the dataflow optimization.

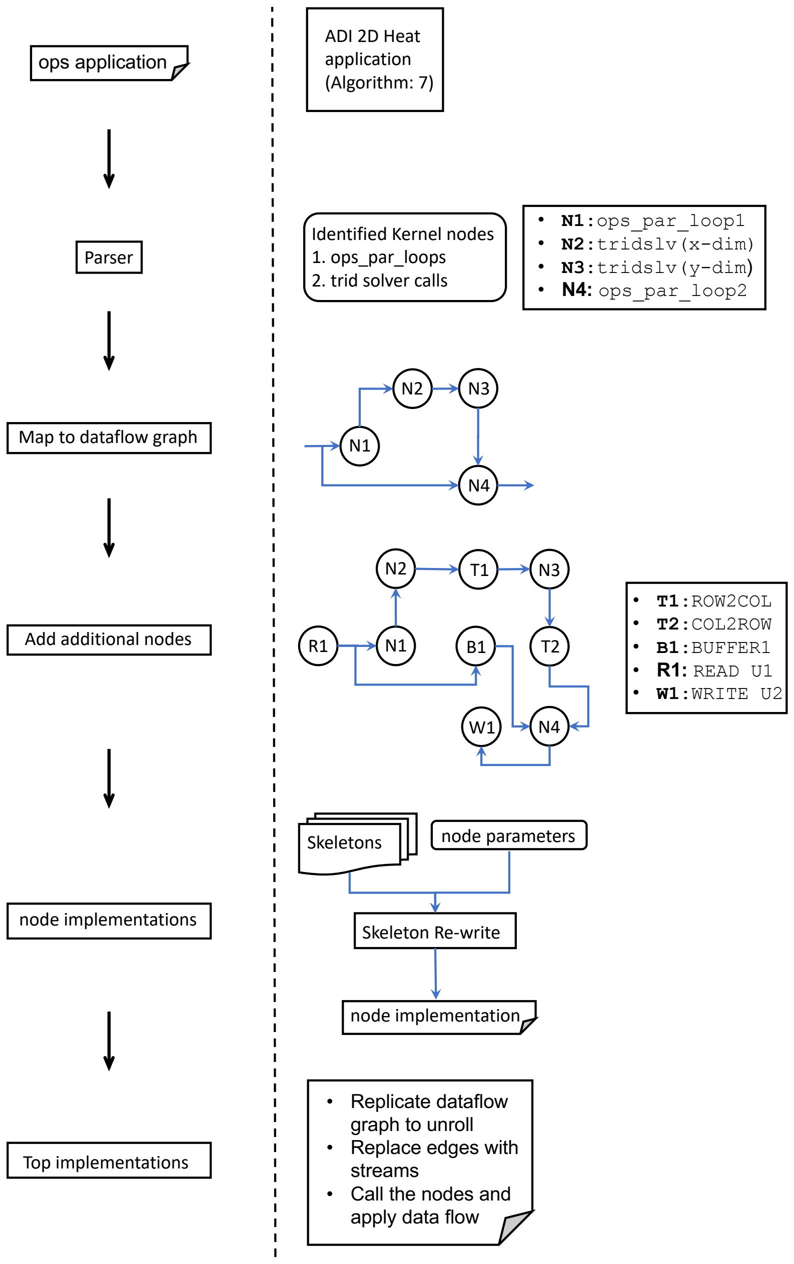


Figure 6.3: Skeleton based source to source translation for FPGA

Table 6.1: Stencil kernel parameters.

Parameter	Symbol
Number of Dimension	`N_DIM`
mesh Dimension	`Dims[N_DIM]`
stencil update start	`SRange[N_DIM]`
stencil update end	`ERange[N_DIM]`
Stencil Order	`S_O[N_DIM]`
Number of stencil Points	`N_SPTS`
Stencil Points	`Spts[N_SPTS][N_DIM]`
Data Type	`Dtype`

6.2.1 `ops_par_loop` nodes: Skeleton For Baseline Design

To transform an `ops_par_loop` into an FPGA target code, the kernel parameters must first be identified from the loop. The essential parameters necessary for implementing a stencil loop specified through `ops_par_loop` on FPGA are outlined in Table 6.1. The number of dimensions of the mesh, `N_DIM` can be obtained from the 3^{rd} argument (it is `two` in Listing 13) of `ops_par_loops`. Dimension of each mesh including boundaries on both sides can be obtained from the `ops_decl_dat` declaration of mesh. The i^{th} dimension will be $size[i] + halo\_pos[i] + halo\_neg[i]$ for the `dat1` in Listing 11. The range of mesh points updated by the stencil kernel is provided through `range` parameter of `ops_par_loop` API. `SRange[i]` and `ERange[i]` in Table 6.1 corresponds 4^{th} parameter of `ops_par_loop`, `range[2*i]` and `range[2*i+1]` respectively in Listing 13. The data type of the mesh points also obtained from 8^{th} argument of `ops_decl_dat`. The Stencil specification of each mesh can be obtained from `ops_arg` object which is passed to `ops_par_loop` as a parameter. The corresponding stencil declaration will detail the stencil dimension, the number of stencil points and the array name corresponding to each stencil point. Stencil points can also be implicitly obtained from memory access of the kernel function specified in `ops_par_loop`.

A baseline design (as discussed in Section 3.1) can be developed once the parameters in Table 6.1 is identified. In order to automate the implementation of the design, a skeleton (Listing 14) has been devised based on commonly employed blocks in baseline FPGA implementations. Firstly, parameters of the function in this skeleton will corresponds to hardware module's interface during high level synthesis. As such, skeleton move the data through the `hls_stream<Dtype>` stream, which is equivalent moving data through FIFOs in the baseline design. Hence, each `ops_arg_dat` in the `ops_par_loop` API will have corresponding `hls_stream<Dtype>` type function parameter in the high level implementation. Design parameters such as mesh dimensions is passed through a `struct` in the skeleton, which would be mapped to AXI lite (a lightweight memory-mapped protocol to configure registers and memory in hardware) interface.

Once the interface of the baseline design is set, stencil computation need to be carried

```
void stencil_kernel(hls::stream<Dtype> &dat0, ..., struct meshParams mp){
    // B1: local variables to hold stencil points
    // B2: local array declaration for window buffers
    // B3: pragma to select window buffer memory type
    // B4: loop invariant declarations, computations

    B5: for(ap_uint<D_SIZE> itr = 0; itr < loopBound; itr++){
        #pragma HLS pipeline II=1
        // B6: mesh point indices calculations
        // B7: for each dat read
            // B7.1: window buffer pointers calculations
            // B7.2: window buffer implementation
            // B7.3: conditional FIFO pop
        // B8: stencil kernel computation

        // B9: for each dat written conditional FIFO push
    }

}
```

Listing 14: Skeleton for the baseline design.

out using the mesh data comes through `hls_stream<Dtype>`. It requires caching/buffering required mesh-points specified by the stencil to update the mesh-points in the next time step. In order to loop through all the mesh-points, a `for` loop (block **B5** in skeleton) is used. Baseline design use window buffers as full data reuse caches, which are implemented as cyclic buffers using on chip memory. A cyclic buffer can be implemented by reading and writing an array at specific distance using HLS. Hence, it would require declaration of arrays (block **B2**) and blocks (**B7.1-7.2**) for access pointer computation and read/write access. Skeleton utilize the block **B3** for choosing the right memory block in the device for a window buffer. The data read from the window buffers will be placed in registers which hold the values specified by stencil. Hence a declaration of variables (block **B1**) for stencil registers is required. The stencil computation arithmetic block (**B8**) access the stencil variables and do the computation. Stencil computation enforce boundary condition which requires indices of the mesh-point which is computed in block **B6**. The values in the stencil registers would be invalid for certain number of iterations until window buffers are fully filled. This would require conditional stream write (block **B9**) block to avoid invalid outputs. Similarly, input stream read should not be attempted (block **B7.3**) in the last set of iteration which account for flushing the data from window buffers. Computation of the conditions, loopbound is carried out in block **B4**.

Following subsections illustrates how to derive the blocks **B1-B9** in detail:

B1 - Local Variables to hold Stencil Points

In the baseline design, registers are employed to store the values indicated by stencils for immediate access by a stencil computation block (see Figure 3.2). The variables in

C++ can be mapped to registers. To simplify the mapping between memory access (`a[OPS_ACC0(0,0)]` in Listing 13) in the stencil kernel specified in `ops_par_loop`, stencil coordinates are used when naming the registers as follows:

$$u(n1, n2, ...) = s\_datID\_Xn1\_Xn2....$$

where `datID` is the ID of corresponding `ops_dat` and X is n, p based on the positivity or negativity of stencil coordinate corresponding values in $\{n1, n2, ..\}$. X will be nill if the corresponding coordinate is zero. As such, registers for the five-point stencil in Listing 13 can be declared as follows:

```
Dtype s_dat0_0_0, s_dat0_n1_0, s_dat0_p1_0, s_dat0_0_n1, s_dat0_0_p1;
```

B2 - Local Array Declaration for Window Buffers

In the Baseline design (refer to section 3.1), window buffers are utilized as a customized cache to achieve full data reuse. Implementing a window buffer often requires multiple on-chip memories. In HLS, static arrays serve as the corresponding element for on-chip memory. The size of the window buffer corresponds to the number of elements between two stencil points, which can be calculated from the mesh dimension and stencil points, as detailed in section 3.1.1. As the baseline design supports dynamically sized meshes, the window buffer size should be set such that it can support the largest mesh size in the given set. Consequently, the window buffer would require multiple memory declarations. The naming convention for window buffers is `window_datID_S1_A_S2`, here `S1` and S2 are corresponding stencil points. Corresponding window buffer declarations for stencil in Listing 13 are:

```
Dtype window_dat0_0_p1_A_p1_0[D_MAX], window_dat0_n1_0_A_0_n1[D_MAX];
```

B3 - `PRAGMA` to Select Window Buffer Memory Type

The required memory size for a window buffer is dependent on the mesh dimension and stencil point pattern. Consequently, the memory requirement could be larger when buffering planes in 3D applications and relatively smaller when buffering lines. In FPGAs , on-chip memory blocks come in two variations: smaller blocks (BRAM, MLAB) and larger blocks (URAM, M20K). When there is a large memory requirement, it is efficient to use a larger block memory than multiple smaller blocks as it would improve FPGA placement and routing. A threshold can be set to select between these memory blocks. `#pragma` as in Listing 15 can be applied to select the local memory type in C++ for Vivado. The latency specifies the number of clock cycles for local memory reads and writes, essentially pipeline stages that improve the clock frequency.

```
#pragma HLS RESOURCE variable=window_dat0_0_p1_A_p1_0 \
core=XPM_MEMORY uram latency=2
#pragma HLS RESOURCE variable=window_dat0_n1_0_A_0_n1 \
core=XPM_MEMORY uram latency=2
```

Listing 15: `#pragma` to select on chip memory block.

B4 - Loop Invariant Declarations, Computations

As the main loop (*B5* in Listing 14) to iterate over the mesh points is fully pipelined, the loop invariant could be calculated sequentially outside the loop to reduce resource consumption. Such loop invariants are a number of `FIFO_POPs`, `FIFO_PUSH_DELAY` and the `LOOP_BOUND`. The following expressions can be used to compute these invariants and we prefer the calculation of these expressions to be implemented on FPGA to support mesh sizes dynamically. Expression for `prime_itr` is presented later in this Chapter in Equation 6.4.

$$\begin{aligned} FIFO\_POPs &= \prod_{i=0}^{N\_DIM} Dims[i] \\ LOOP\_BOUND &= \prod_{i=0}^{N\_DIM} Dims[i] + prime\_itr \\ FIFO\_PUSH\_DELAY &= prime\_itr \end{aligned}$$

We note here, Vivado-HLS supports custom-width integers. Smaller width integer operation requires less FPGA area and will help to achieve better frequency. As such required minimum bit-width can be calculated as follows.

- Two's compliment addition's output requires one-bit width more place holder than the maximum bit width of operands
- Two's compliment multiplication's output requires the sum of the bit width of operands for the output placeholder

As such, loop invariants can be declared and defined as follows.

```
ap_uint<SIZE_FIFO_POPs> fifoPops = FIFO_POPs ;
ap_uint<SIZE_LOOP_BOUND> loopBound = LOOP_BOUND;
ap_uint<SIZE_FIFO_PUSH_DELAY> fifoPushDelay = FIFO_PUSH_DELAY;
```

B5 - Flattened For Loop

In order to obtain performance and area-saving benefits, the skeleton employs a flattened loop instead of a nested loop. Additionally, the flattened loop construct remains consistent across all stencil applications, except for the loop iteration variable width `itr`. To ensure that each loop iteration is executed every clock cycle, the directive `#pragma pipeline II=1` is used.

B6 - Mesh Point Indices Calculation

To enforce the boundary conditions in stencil computations, it is necessary to determine the mesh indices. These indices can be obtained by utilizing the iteration value of the loop. The computation of mesh indices can be performed as in Listing 16. In this context,

```
ap_uint<D_SIZE> CIndex[N_DIM];
B5: for(ap_uint<D_SIZE> itr = 0; itr < totalCount; itr++){
    CIndex[0] = itr % Dims[0];
    CIndex[1] = (itr / (Dims[0]) % Dims[1];
    CIndex[2] = (itr / (Dims[1]*Dims[0]) % Dims[2];
    ...
}
```

Listing 16: Mesh point index computation.

the array `CIndex[]` stores the mesh cell indices across the dimensions for each iteration. For an n-dimensional mesh, this implementation can be extended to the k^{th} dimension, where $k < n$, by utilizing the following expression:

$$ind\_k = \frac{itr}{\prod_{i=0}^{k-1} Dims[i]} \% Dims[k]$$

The direct implementation of the aforementioned expressions is computationally expensive, as it necessitates the use of modulus operators, dividers, and multipliers. However, this calculation can be carried out using counters (implemented with adders in hardware) and comparators, which consume far fewer resources. Additionally, another register block, denoted as `CIndexD[]`, is employed in Listing 17 to improve clock frequency by minimizing the fanout (number of connections it feeds to in the circuit) and critical path on the update of the `CIndexD[]` registers.

```
ap_uint<D_SIZE> CIndex[N_DIM];
ap_uint<D_SIZE> CIndexD[N_DIM];
bool cmp[N_DIM]
B5: for(ap_uint<D_SIZE> itr = 0; itr < loopBound; itr++){
    CIndex[0] = CIndexD[0];
    CIndex[1] = CIndexD[1];
    CIndex[2] = CIndexD[2];
    ....

    bool cmp[0] = (CIndex[0] == Dims[0]-1 );
    if(cmp_0){ CIndexD[0] = 0;} else { CIndexD[0]++;}
    bool cmp[1] = (cmp[0]  && ind_1 == Dims[1] -1);
    if(cmp[1]){CIndexD[1] = 0;} else if(cmp[0]} {CIndexD[1]++;}
    ...
    bool cmp[k] = (cmp[k-1] && CIndex[k] == Dims[k] -1);
    if(cmp[k]){CIndexD[k] = 0;} else if(cmp[k-1]) {CIndexD[k]++;}

}
```

Listing 17: Resource optimized mesh index computation.

B7.1 - Window Buffer Pointers Calculations

Window buffers are designed such that it buffers a certain number of elements, let's say n_b. This could be implemented using on-chip memory by maintaining n_b distance between reads and write locations, a chunk of memory with size n_b can be read and written cyclically. A naive implementation would require two access pointers, one for reading and the other for write. In order to make the implementation simple, we use a single access pointer and make the cyclic distance equal to n_b. A window buffer can be implemented as in Listing 18, here HLS tool will automatically enforce the Write After Read (WAR) dependency.

```
B5: for(ap_uint<D_SIZE> itr = 0; itr < loopBound; itr++){
    ap_uint<D_SIZE> ptr_0 = itr % n_b;
    // Write after Read
    out = windo_0_1[ptr_0];
    windo_0_1[ptr_0] = in;
    ...
}
```

Listing 18: Window buffer index computation.

The value of n_b can be calculated using the steps outlined in section 3.1.1. Additionally, the calculation of `ptr_0` can be optimized using a counter-based implementation, thereby avoiding the expensive implementation of modulus operators on the FPGA. When multiple window buffers require buffering the same number of elements, a single pointer can be utilized to conserve FPGA resources.

B7.2 - Window Buffer Implementation

Multiple window buffers as above can be chained together as in Listing 19 to buffer required number of mesh points between stencil points. Data will move through local variables when stencil points are adjacent.

```
B5: for(ap_uint<D_SIZE> itr = 0; itr < totalCount; itr++){
    ap_uint<D_SIZE> ptr_0 = itr % number_of_elements_buffered_0;
    ap_uint<D_SIZE> ptr_1 = itr % number_of_elements_buffered_1;

    out_1 = windo_0_1[ptr_1];
    windo_0_1[ptr_1] = in_1; // Write after Read

    // data moves through registers
    in_1 = var_0;
    var_0 = out_0;

    out_0 = windo_0_1[ptr_0];
    windo_0_1[ptr_0] = in_0; // Write after Read
    ...
}
```

Listing 19: Window buffer implementation.

B7.3 - Conditional FIFO Pop

Using blocking FIFO operations such as FIFO pop and FIFO push can simplify the design and implementation of the stencil loop on FPGA. However, the number of read attempts should precisely match the amount of data available through that FIFO, otherwise, the design will hang due to blocking FIFO operations. As the flattened loop's bound is modified to incorporate prime time, FIFO reads should not be attempted during the last few iterations. Conditional guards can be utilized to implement this, as shown in Listing 20.

```
B5: for(ap_uint<D_SIZE> itr = 0; itr < totalCount; itr++){
    if(itr < fifoPops){
        in_0 = dat_0.read();
    }
    ...
}
```

Listing 20: Conditional FIFO pop.

`fifoPops` will be equal to the number of mesh cells. This count would be calculated in *B4: loop invariant declarations, computations*

B8 - Stencil Kernel Computation

This block corresponds to a kernel function that is defined in the `ops_par_loop`. In this kernel function, memory accesses must be substituted with stencil points that are stored in registers. To perform this substitution, we will apply a direct mapping described in *B3: local variables to hold stencil points*. We should ensure that mesh points beyond the specified `iter_range` in `ops_par_loop` remain unaffected. this would require a conditional update similar to kernel implementation on GPU. As such **B8** block for kernel in Listing 13 will as in Listing 21.

```
Dtype result = (s_n1_0  +s_p1_0 + s_0_n1 + s_0_p1)*0.5f + \
            s_0_0 * 0.5f;
bool cond = (CIndex[0] < SRange[0] || CIndex[0]  > ERange[0] \
|| CIndex[1] < SRange[1] || CIndex[1]  > ERange[1])
Dtype out = cond ? s_0_0: result;
```

Listing 21: Stencil computation using values in the register.

B9 - Conditional FIFO Push

As stencil computation is done before window buffers not being filled enough is invalid. The minimum number of iterations required to fill the window buffers sufficiently is represented by the variable `prime_itr`. The condition for pushing the result to the output `hls::stream<Dtype>` is illustrated in Listing 22.

```
B5: for(ap_uint<D_SIZE> itr = 0; itr < totalCount; itr++){
    if(itr > prime_itr){
        dat_1 << out_0;
    }
    ...
}
```

Listing 22: Conditional FIFO push

Above presented skeleton based transformation of `ops_par_loop` requires further improvements for exploiting available compute capability of FPGAs (vectorization) and support range of mesh sizes effectively (batching and spatial blocking). Following sections illustrates required modification in the blocks **B1-B9** transformation to enable those optimizations.

6.2.2 `ops_par_loop` nodes: Vectorization

In the current implementation, only one mesh point is updated per iteration or clock cycle. However, it is possible to update multiple mesh points in a single iteration to improve the performance. To enable vectorization, certain blocks (B1 to B9) in the existing skeleton need to be modified. The optimization for vectorization also necessitates a wider data path from the input stream to the output stream, as illustrated in Figure 6.4. This can be achieved by utilizing a struct data type that includes a static array of elements or by using a wider integer datatype as in Listing 23.

```
// struct based implementation
struct vecBlockS{
    DType points[V];
};

// wider integer based
typedef ap_uint<V*sizeof(DType)> vecBlockI;
```

Listing 23: Wider data types.

The use of `struct` based wider data type enable to implement vectored computation as mesh elements can be easily indexed. The `struct vecBlockS` in Listing 23 can also be utilized to create wider on-chip memories. However, older HLS compilers have a tendency to generate separate RAM blocks for each array element inside the struct, which can lead to under utilization of URAMs if the size of the array element is less than the URAM's width. To address this issue, later HLS compilers concatenate the array elements and map them to a wider port memory. A similar optimization can also be performed in older HLS compilers by declaring wider integer types. A portion of the wider integer can be accessed through range (`vecBlockI.range(a,b)` for the wider datatype in Listing 23), and a union construct can be used to reinterpret it to the desired type.

In order to obtain better external memory throughput, vectorization is preferred along

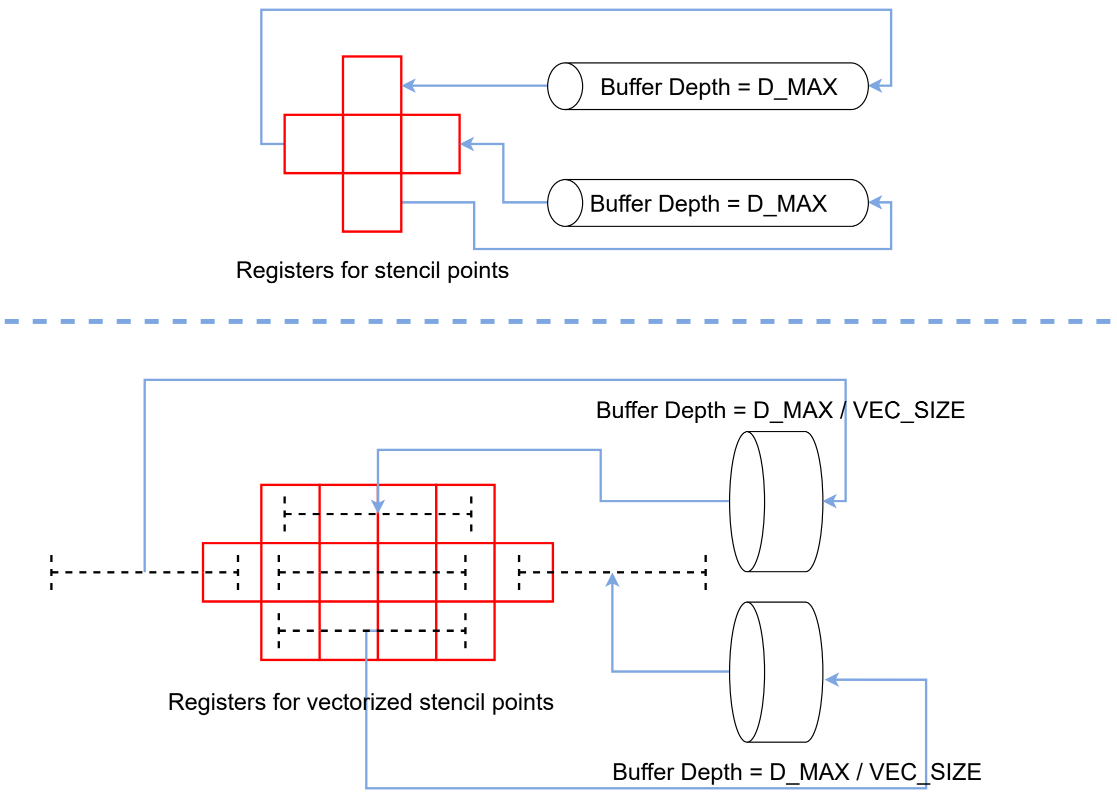

Buffer Depth = D_MAX
Buffer Depth = D_MAX
Registers for stencil points
Buffer Depth = D_MAX / VEC_SIZE
Registers for vectorized stencil points
Buffer Depth = D_MAX / VEC_SIZE

Figure 6.4: Data path for vectored stencil computation

the first dimension since mesh cells are then located in consecutive memory locations. It is important to note that vectorization optimization assumes that the first dimension (`Dims[0]`) is a multiple of the vectorization factor, denoted as V. To simplify the data path implementation using the existing skeleton, stacked vector stencil points can be treated as a single stencil. This stacked stencil can then be segmented into vector blocks, which we refer to as vector stencils. The number of vector blocks between two points in a vector stencil can be determined in a similar manner to standard stencils. Therefore, in order to optimize for vectorization, the above skeleton blocks require the following modifications.

- **B1, B3:** data type (`Dtype`) should be wider data type such as `vecBlockS` to hold vector block. The corresponding array size should be $1/V$ as each word of RAM contains V number of mesh cells.

```
struct vecBlockS window_0_1[D_MAX/V];
```

- In calculations such as `loopBound`, `fifo_Pops`, mesh indices, window buffer Pointers and `prime_itr`, the first dimension `Dims[0]` should be assumed as $Dims[0]/V$.

- **B8:** Stencil kernel computation should be vectored. This can be implemented using a fully unrolled for loop as in Listing 24.

```
for(int v = 0; v < V; v++){
    #pragma HLS unroll
    ap_uint<D_SIZE> ind0 = CIndex[0] * V + v;
    Dtype s_dat0_n1_0_scalar, s_dat0_p1_0_scalar;
    s_dat0_n1_0_scalar  = (v == 0) ? s_dat0_n1_0.data[V-1] : s_dat0_0_0[v-1];
    s_dat0_p1_0_scalar  = (v == V-1) ? s_dat0_p1_0.data[0] : s_0_0[v+1];

    Dtype result = s_dat0_n1_0_scalar  + s_dat0_p1_0_scalar + \
        s_dat0_0_n1.data[v] + s_dat0_0_p1.data[v] + s_dat0_0_0.data[v];
    bool cond = (ind0 < SRange[0] || ind0  > ERange[0] \
    || CIndex[1] < SRange[1] || CIndex[1]  > ERange[1])
    Dtype out.data[v] = cond ? s_dat0_0_0.data[v]: result;
}
```

Listing 24: Vectored stencil computation.

Parameter	Data Structure
Tile Block Sizes	`Tiles[]`
Tile Block Indexs	`TileID[]`
Mesh cell index in A Tile Block	`LocalID[]`
global mesh cell index	`globalID[]`
Halo Region size in each Dimension	`HO[]`

Table 6.2: Spatially Blocked Design parameters.

6.2.3 ops_par_loop nodes: Batched Computation

The batching optimization presented in section 3.3.3 improves throughput specifically for medium and small meshes. Batching on the last dimension is preferred because it doesn't require additional on-chip memory and amortizes pipeline latency as `prime_itr` is only present for the first mesh. Batching on the last dimension can be viewed as extending the last dimension by a batch size of B times. However, the boundary condition for each mesh should be applied and it requires index calculation. To simplify this, N_DIM will increase to $N\_DIM + 1$ in all calculations (last dimension becoming batch index), except in **B8** of the stencil computation.

6.2.4 ops_par_loop nodes: Spatially Blocked Computation

The optimization technique referred to as spatially blocked optimization to support larger meshes, as explained in section 3.3.1, involves dividing the mesh into smaller blocks to allow for effective computation using the available on-chip memory. The skeleton of the kernel for this optimization approach is similar to that of the baseline design, but with minor adjustments made to blocks **B1 -B9**. In this case, the tile dimensions (`Tiles[]`) should be viewed as the mesh dimensions in the baseline design. It is also necessary to have actual mesh indexes in order to identify the mesh boundary and enforce boundary conditions. These global mesh indexes will be utilized as a condition for updates in block **B8**. Furthermore, the identification of the global index of the mesh cell requires the tile block's `TileID[]` positions in the entire tile blocks. Assuming uniform spatial blocking with the same tile sizes, the mesh cell index for the i^{th} dimension can be expressed as $\texttt{TileID[i]} \times (\texttt{Tile[i]} - 2 \times HO[i]) + \texttt{LocalID[i]}$.

Since there are multiple tile blocks that need to be processed, it requires another control structure to loop through all blocks. This can be simply implemented by calling a kernel function with required parameters inside a loop structure as in Listing 25.

```
for(ap_uint<D_size> blk; blk < totalBlocks; blk++){
    // Tile block offset from memory or FIFO
    stencil_kernel(dat0, ..., meshParams);
}
```

Listing 25: Looping through spatial blocks.

This implementation is simple and skeleton for baseline design can be used with minor modifications for **`stencil_kernel`** function. The benefit of this approach is that each tile block undergoes independent processing, thereby enabling support for variable tile sizes. This feature is particularly advantageous when smaller tiles suffice at the mesh's corners. To accomplish this, an added data structure indicating the offset of each tile is necessary. The host program can provide this information through either a memory or stream interface, and the corresponding offset can be transferred to the kernel.

The drawback is that when dealing with tile blocks on higher dimensional meshes, the **`prime_itr`** latency is typically substantial in comparison to the complete processing time of a tile block. This is because the dimensions of the tile could shrink due to on-chip memory constraints, while the number of mesh points that necessitate buffering before initiating computation increases when moving to higher dimensional meshes. Once again, the solution to this issue is batching, where tile blocks are batched instead of different meshes. The computation of the total number of batches is depend on the size of the overlapping or "halo" region. The halo region's size depends on the stencil order along a dimension and expands when multiple **`ops_par_loops`** are chained, as invalid computation propagates. Let us assume that the halo region along each dimension of the tile blocks is denoted by `HO[]`. As a result, the number of tile blocks, N_Tiles, will be calculated as follows, assuming uniform tile blocks:

$$N\_Tiles = \prod_{i=0}^{N\_DIM-2} \left\lceil \frac{Dims[i]}{Tile[i] - 2 * HO[i]} \right\rceil$$

Inspired by the advantages of batched computation of tile blocks, we prefer utilizing the same skeleton for batched optimization, albeit with some modifications. The key difference here is that global mesh indices must be determined based on `TileID[]`. The computation of `TileID[]` can be accomplished in a similar manner to the computation of mesh indexes in the baseline design. As we are computing both `TileID[]` and `LocalID[]` from the flattened loop iteration, we can regard the dimensions of `TileID[]` as an extension of the Tile block dimensions.

- **B4: loop invariant declarations, computations**
  Compared to the skeleton for batched optimization, loop in-variants like `loopBound` and `fifoPops` need to be computed in different way. In this scenario, the last dimension of the tile block should be regarded as expanded by a factor of N_Tiles.

- **B6: Mesh point indices calculations**
  The indices within the tile block (`LocalID`) and the tile block index (`TileID[]`) can be calculated in the typical fashion by treating each tile block dimension as an extension of the mesh dimensions. Once the tile block index (`TileID[]`) and the index within the tile block (`LocalID[]`) have been calculated, the global mesh point index (`globalID[]`) can be determined.

6.2.5 Global Memory Access Nodes

The source and sink nodes of the dataflow graph for global memory access serve as a means of separating communication from computation. This separation allows for effective optimization and simplified bottleneck analysis. Global memory access nodes issue requests for data from external or nearby memory via AXI interfaces. The AXI protocol is a widely used protocol for memory-mapped data movement within a chip. To initiate memory transfers through the AXI interface using the AXI protocol, a Global memory access node includes an AXI controller block. The AXI interface is connected to the external memory controller, which then translates AXI memory requests to physical memory-specific protocol and manages the data movement.

The global memory access nodes play a critical role in determining the performance of the dataflow graph pipeline. If the throughput of a global memory access node is lower than the compute pipeline connected to it, the entire pipeline will stall. Improving the external memory throughput depends on various factors, including design parameters and physical memory and memory controller specifications. Therefore, the rate at which data is transferred between global memory access nodes and external or nearby memory depends on physical memory specification, memory controller, AXI bus architecture and AXI controller.

The achievable memory throughput between access nodes and external memory will be minimum of the physical memory throughput, memory controller throughput, and AXI bus throughput. Once the required throughput for a global memory access node is determined, the number of memory banks and memory controllers required can be identified. If a larger throughput is needed for a specific data structure, it can be divided into cyclically assigned chunks to dedicated memory banks.

The AXI interface parameters can be chosen to match the throughput of the memory controller and memory banks. The maximum theoretical throughput of the AXI bus can be calculated based on the data width and operating frequency of the AXI interface as in Equation 6.1, ignoring AXI transaction latency. Here, f is clock frequency and W is width of AXI port.

$$BW = 2 * f * W \tag{6.1}$$

Multiple data structures can share a single bank if the required throughput for a single data structure is much lower than the provided throughput of that memory bank. The kernel's data structure can be assigned to a memory bank through the AXI port provided by the HLS tool using the `pragma HLS INTERFACE` (Listing 26) in the top function of the kernel (`kernel_0`). The AXI width corresponds to the size of the data type of the global memory pointer, and the AXI burst length is derived by the HLS tool based on the memory access pattern. The same `pragma` (Listing 26) can be used to specify additional AXI parameters such as maximum outstanding transactions and maximum burst length that need to be supported by the AXI bus interface.

```
extern "C" {
void kernel_0(
    DType*  g_data0,
    DType*  g_data1,
    ....  // other parmeters
        ){
    #pragma HLS INTERFACE m_axi port=g_data0 offset = slave \
    bundle = gmem0 max_read_burst_length=64 max_write_burst_length=64 \
    num_read_outstanding=4 num_write_outstanding=4
    .....
}
```

Listing 26: Memory interface configuration.

The purpose of this section is to classify memory access patterns that are typically found in the global memory access kernels of structured-mesh based applications, and to present skeletons for implementing these kernels. Three common memory access patterns are identified, namely sequential memory access, tiled memory access, and accessing memory at specific distances. Depending on the memory access pattern, specific loop transformations and interface configurations are necessary to optimize memory throughput.

Sequential memory access nodes

Sequential memory access is common in baseline and batched design where mesh points are read sequentially one after another in memory. In HLS, a loop can be used to run through the mesh and obtain each cell from external memory. A flattened loop with sequential memory access is preferred as this will correspond to a single memory request. This request will later be split into multiple AXI transactions by the AXI controller based on a maximum AXI burst length limitation of 256 or 4K total bytes limitation for single transaction. Since burst length can be set to the possible maximum value, it won't require many outstanding transactions to get better memory throughput.

If a `while` loop is used, then it will be mapped to individual transactions with burst length size one, this is because usually exit condition of while loop can't be resolved at compile time. This will cost in performance as each AXI transaction can take multiple tens of clocks. If transactions are performed one after another, it will be very inefficient. If multiple outstanding transactions are set then, it will cost additional FPGA resources and make FPGA implementation complex. Hence a flattened `for` loop is preferred for sequential memory access. Moreover, coalesced memory access is preferred as it will help to achieve better memory bandwidth. A struct data type with fixed size array or wider integer type can be used for this purpose. Skeleton for preferred `for loop` implementation for sequential memory access is as in Listing 27 (similar structure for global memory write).

```
struct Dtype {float data[4]};
static void gMem_access_node_0(Dtype* g_data0, hls<float>::stream out_s,
    int total_length){
    for(ap_uint<D_size> t_0; t_0 < total_length; t_0++){
        #pragma HLS PIPELINE II=4
        Dtype data_w = g_data0[t_0]; // wider memory access
        // pushing individual data to output stream
        out_s << data_w.data[0];
        out_s << data_w.data[1];
        out_s << data_w.data[2];
        out_s << data_w.data[3];
    }
}
```

Listing 27: Coalesced memory access.

Tiled memory access nodes

Spatially blocking optimization (section 3.3.1) requires reading chunks of data at specific distances. This can be effectively implemented using a nested loop with depth two as in Listing 28. Here, inner loop count corresponding to size of the chunks to be read and `Ap.total_t_x` corresponds to number of such chunks. `# pragma HLS PIPELINE` is applied to inner loop to advice HLS tool to infer burst mode AXI transfers. As AXI transactions is associated with significant latency (lets say l_axi clocks) and throughput can be calculated as in Equation 6.2, here len_{burst} which corresponds to chunk size.

$$BW = 2 * f * \frac{len_{burst}}{len_{burst} + l_{axi}} \tag{6.2}$$

In that case, we prefer to set higher number of outstanding transaction to hide latency of previous transactions. If the number of outstanding transaction is N_{trans}, corresponding bandwidth will be as in Equation 6.3. Since outstanding transaction will require additional onchip memory for buffering, we prefer to set N_{trans} such that over 90% bandwidth provided by memory bank is achieved.

$$BW = 2 * f * \frac{N_{trans} \times len_{burst}}{N_{trans} \times len_{burst} + l_{axi}} \tag{6.3}$$

In Listing 28 `offset` need to be calculated for each chunks and it could be calculated using the parent loop iteration, similar to computing the mesh index from the flattened loop iterations in baseline design skeleton.

```
static void gMemT_access_node_0(Dtype* g_data0, hls<float>::stream out_s,
    struct AppParams Ap){

for(ap_uint<D_size> t_x = 0; t_x < Ap.total_t_x; t_x++){
    // Tile block Id computations
    // local ID computation except the first dimension
    // Address offset computation

    // Single AXI request - Burst mode transfer
    for(ap_uint<D_size>  i = 0; i < Ap.Tile_X; i++){
        # pragma HLS PIPELINE II=1
        Dtype data = g_data0[offset+i];
    }
}
}
```

Listing 28: Tiled memory access.

6.2.6 Delay Buffers Nodes

Branches could arise when mapping the OPS API calls to the dataflow graph computing and there could be circular branches as in Figure 6.5, when there are at least two paths from one node to another. As blocking FIFO operations are used in kernels, it could lead to deadlock in circular branches if data pushed on one branch is not consumed, making that FIFO full. Figure 6.5 illustrate such a possible scenario. Here `kernel:1` flushes the data along path A and B. Let's assume d_b number of elements need to be buffered (E.g window buffers) before the first output to be released from `kernel:2` to path C (d_b is equal to $prime\_itr$). `kernel:3` would require a data from each path B and C to start computation. As data coming from path C is delayed, path B become full if the buffer capacity is exceeded. Now `kernel:1` can't release the data to path A as well, as data should be pushed to paths A and B simultaneously. This essentially creates a deadlock in the data flow graph.

In some cases, `kernel:2` doesn't require a certain number of elements to be buffered but each element entering the kernel is released after a certain number of clocks due to pipeline stages S in `kernel:2`. In this case, there won't be a deadlock but intermittent stalls if the buffer capacity of path B is not enough. In order to have the optimal operation of the above dataflow graph branch, the buffer capacity of the FIFO/Stream should be higher than $d_b + S$.

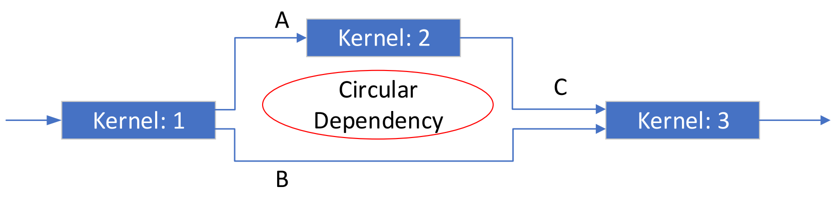


Figure 6.5: Circular dependency on branches.

Value d_b of a kernel can be calculated using the stencil provided in the `ops_par_loop` for the particular data structure. This is equivalent to the number of elements between the stencil's starting point's index (`I_s`) and updating point's index (`I_u`) as follows.

$$\begin{aligned} prime\_itr = d_b &= (Spts[I\_u][0] - Spts[I\_s][0]) \\ &+ \prod_{i=1}^{Dims-1} (Spts[I\_u][i] - Spts[I\_s][i]) * Dims[i] \end{aligned} \tag{6.4}$$

Value S can't be precisely calculated as HLS synthesizer will add additional pipeline stages to meet the target frequency. The number of pipeline stages between the input and the output of a kernel can be obtained by inspecting the HLS scheduler report. it is usually less than a few hundred pipeline stages. This also can be estimated using arithmetic computations over the critical path of the expressions. Additional slack like 100 can be added to this to allow additional pipeline stages introduced by the HLS compiler. Once the total number of elements that need to be buffered is calculated, that will be the depth of the required buffer. A delay buffer node can be implemented using `hls::stream<Dtype>`. It will be mapped to FIFO on hardware which is usually implemented using register/BRAM/URAM.

```
static hls::stream<DType> pathB;
#pragma HLS STREAM variable = pathB depth = FIFO_DEPTH
```

6.2.7 `ops_tridMultiDimBatch`: Tridiagonal Solver Nodes

Tridiagonal systems are common in structured mesh-based implicit applications and OPS provides the following API for solving tridiagonal systems formed along each dimension of multi-dimensional meshes.

```
void ops_tridMultiDimBatch(int ndim, int solvedim, int* dims, ops_dat a, \
    ops_dat b, ops_dat c, ops_dat d, ops_tridsolver_params *tridsolver_ctx)
```

This API supports solving multi-dimensional tridiagonal systems along each dimension of 1D,2D and 3D meshes. Input tridiagonal matrix coefficients (`a`,`b`,`c`), RHS (`d`) and output (`d`) are multi-dimensional meshes. This API is mapped to highly optimized CPU and GPU library functions when generating target implementation for CPUs and GPUs.

We also employ a similar approach using library functions along with specializations for the cases where tridiagonal matrix coefficients are constants (e.g. Poisson equation) or could be internally calculated. In those cases, the coefficient computation can be fused with the library function to save on-chip memory required for coefficient interleaving as well as required off-chip memory space and bandwidth for those coefficients. Again this is also a skeleton-based technique but just a few lines related to setting coefficients need to be replaced.

In contrast to CPU and GPU tridiagonal system libraries that OPS translator employs, where data move from/to global memory, our batched tridiagonal solver library presented in Chapter 4 gets data from and to streaming interfaces as dataflow graph nodes are connected using streams. This requires certain data re-organization transformations such as rows to columns transpose in order to feed data to the tridiagonal solver in the required order (`data re-organiser nodes`). Listing 29 illustrates how library modules are used to make such data re-organization transformation for solving along the first dimension. In essence, these library functions are templated, taking vectorization factor, dimension and data type as the template parameters. First routines (`interleaved_row_blockV`) take V number of systems and stream out $V \times V$ blocks and these $V \times V$ blocks are fed to $V \times V$ transpose module (`stream_VxVtranspose`) to feed the output to V number of Thomas solver pipelines as illustrated in Figure 4.1. The last two library functions re-organize the data and send the output of each system sequentially. for simplicity, we have presented the case where coefficients can be internally generated, if not, coefficient inputs (a, b, c) also should go through such transformations.

```
interleaved_row_blockV<V, 128>(d_stm_0[1], d_stm_0[2], M, N, B, 1);
stream_VxVtranspose<V, float>(d_stm_0[2], d_stm_0[3], M, N, B, 1);
thomas_interleave<V, float, 128>(d_stm_0[3], d_fw_stm[0], M, B_X, ReadLimit_X);
thomas_forward<V, float, 128>(d_fw_stm[0], c2_fw_stm[0], d2_fw_stm[0], M, B_X);
thomas_backward<V, float, 128>(c2_fw_stm[0], d2_fw_stm[0], u_stm_0[0], M, B_X, \
    ReadLimit_X);
stream_VxVtranspose<V, float>(u_stm_0[0], u_stm_0[1], M, N, B, 1);
undo_interleaved_row_blockV<V, 128>(u_stm_0[1], u_stm_0[2], M, N, B, 1);
```

Listing 29: `tridslv(x-dim)` on FPGA.

On the other hand, solving along the second dimension of mesh requires a plane transpose if input data is organized sequentially along the first dimensions. it will be mapped to the following library functions as in Listing 30. Solving along the third dimension

```
row2col<V, 128>(u_stm_0[2], d_stm_0[4], M, N, B);
thomas_interleave<V, float, 128>(d_stm_0[4], d_fw_stm[1], N, B_Y, ReadLimit_Y);
thomas_forward<V, float, 128>(d_fw_stm[1], c2_fw_stm[1], d2_fw_stm[1], N, B_Y);
thomas_backward<V, float, 128>(c2_fw_stm[1], d2_fw_stm[1], u_stm_0[3], \
    N, B_Y, ReadLimit_Y);
col2row<V, 128>(u_stm_0[3], u_stm_0[4], M, N, B);
```

Listing 30: `tridslv(y-dim)` on FPGA.

requires the support of the off-chip memory as buffering the whole 3D mesh using on-chip memory is not feasible. Hence it requires another global memory access node to be added and which will buffer each mesh on off-chip memory and read planes formed using 0^{th} dimension and 2^{nd} dimension. Again a transpose library function should be utilized to feed the systems along 3^{rd} dimension.

6.2.8 Building Dataflow Graph

Once required nodes are implemented using the above methodology, top-level dataflow can be built by replacing the edges of the dataflow graph with the stream and applying the dataflow optimization over the node computation calls as in Listing 31. Additionally kernel arguments need to be computed above the dataflow region.

```
void ops2dataflow(Dtype* g_data0, Dtype* g_data1, ..., struct meshParams mp){
// hls::stream<Dtype> declarations
// kernel argument computations
#pragma HLS dataflow
// all the nodes in the dataflow graph
}
```

Listing 31: Dataflow optimization.

6.3 Optimal Design Parameter Identification

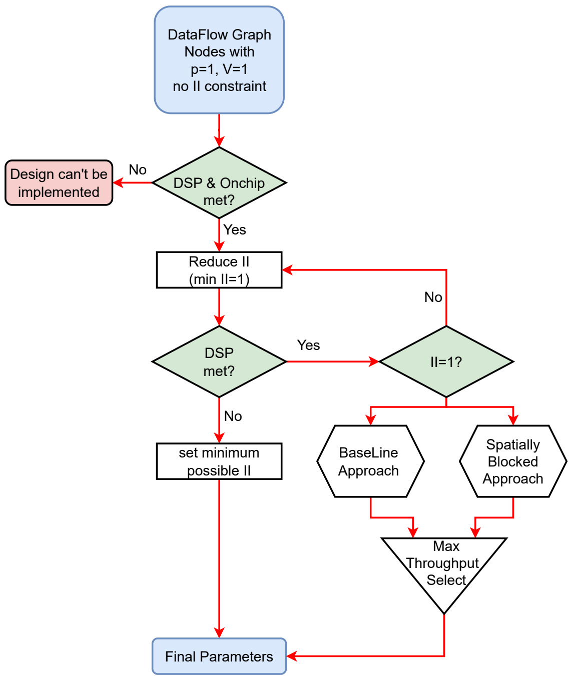


Figure 6.6: High-level overview to estimate design parameters.

Each node in the dataflow graph requires a certain amount of FPGA resources (LUT, registers, on-chip memory, DSP, etc) and FPGAs come with a fixed amount of these resources. Optimizations such as pipelining loops and vectorization will scale up the resource consumption, indeed resource utilization can be estimated based on these optimization parameters. In order to enable these optimisations and choose design parameters, the resource consumption of the dataflow graph needs to be estimated, to make sure overall resource consumption is within the available resource limit. FPGA device resource consumption of a dataflow graph can be estimated in two ways. Resources such as DSP units and on-chip memory consumption can be estimated theoretically but other device resources such as LUTs and registers can be better estimated using the Vitis HLS synthesis report. In order to get the Vitis HLS report, the dataflow graph needs to be implemented using the FPGA target language. Based on previous transformation techniques, when design parameters are known, the kernel can be automatically generated.

As each kernel can have a different set of design parameters, the theoretical search space to find optimal parameters is huge. This search space can be narrowed down as uniform data flow requires similar V/II in each node as the lower performance of one node will reduce the whole dataflow graph's throughput. We prefer making $II = 1$ before scaling the V to make the design use fewer resources to get the same performance. If data structures with different numbers of mesh points are used, then V should be proportionate to the number of mesh points, assuming adjusted V for a node is V_i. Let's assume the data container for resource consumption `struct FPGA_resource (DSP, mem, LUT, Reg)` and such resource consumption values for node i for target V of largest output data structure is `RNode(`i, V_i`)`. If the sum of the resource utilization in the nodes is a few times less than available FPGA resources the entire data flow graph can be unrolled p times if such dataflow the graph is inside an iterative loop. As such, we propose the flow chart in Figure 6.6 to find the optimal design parameter for the design. It explores three design parameters (II, V, p) and two design strategies, baseline design and spatially blocked design. The best design strategy and corresponding design parameters are estimated through the algorithm given in Figure 6.6.

In this algorithm, we first check if it is possible to reach II=1 for the dataflow graph, if it is possible we check both the baseline approach and spatially blocked approach to select the approach which gives better performance. Finding the design parameters using the baseline approach and spatially blocked approach are described in Algorithm 8 and Algorithm 9. Here, we define the total available FPGA resources in data structure `Struct FPGA_resource F`. `Thpt(V,p)` and `ThptT(V,p, Tile)` compute the throughput of the dataflow graph for baseline and spatially blocked design respectively. `calTile(p)` computes the maximum possible Tile for the dataflow graph for given p.

In both baseline and spatially blocked approaches, search space is minimized by estimating the maximum possible values for design parameters for V (vectorization factor) and p (iterative loop unroll factor). Since at least one FPGA resource (on-chip memory, DSP) will scale with p and V, the maximum value for p can be obtained when V is smallest, $V = 1$. Aggregate resource consumption for dataflow graph nodes can be

Algorithm 8: `Baseline approach`

```
1: $r0 \leftarrow \sum_{i=0}^{\#kernels} RNode(i,1)$
2: $k0 \leftarrow max(r0.DSP/F.DSP, r0.mem/F.mem, r0.LUT/F.LUT, r0.Reg/F.Reg)$
3: $p\_max \leftarrow 1/k0$
4: $V_{maxRes} \leftarrow 1/k0$
5: $V_{maxBW} \leftarrow \frac{total\_banwdidth}{2 \times f \times sizeof(DType) \times no\_of\_global\_nodes}$
6: $V_{max} \leftarrow min(V_{maxRes}, V_{maxBW})$
7: $V_f \leftarrow 1$
8: $p_f \leftarrow 1$
9: $T_{max} \leftarrow Thpt(1,1)$
10: for $p = 1, 2, ..., p\_max$ do
11:   for $V = 1, 2, ..., v\_max$ do
12:     $r \leftarrow \sum_{i=0}^{\#kernels} RNode(i,V)$
13:     $k \leftarrow max(r.DSP/F.DSP, r.mem/F.mem, r.LUT/F.LUT, r.Reg/F.Reg)$
14:     if $K > 1$ then
15:       continue
16:     end if
17:     $T \leftarrow Thpt(V,p)$
18:     if $T > T_{max}$ then
19:       $T_{max} \leftarrow T$
20:       $V_f \leftarrow V$
21:       $p_F \leftarrow p$
22:     end if
23:   end for
24: end for
```

estimated by setting $V = 1$ and the accumulation resource consumption of each node. At this point, the most used resource will determine the maximum possible iterative unroll factor (p_max) as resource consumption will scale with p. Similarly, V will also scale DSP resource consumption, hence maximum V_{maxRes} can be obtained by setting maximising V until it hits the DSP limit. Other than DSP units, available off chip memory bandwidth also restrict the scaling of V. Required off chip memory bandwidth is proportional to V, number of global memory access nodes, operating frequency (Default frequency 300 MHz can be used for U280) and size of `DType`. We estimate the maximum possible V_{maxBW} from other known factors related to bandwidth requirement. Maximum throughput T_{max} and design parameters can be initialized to ($Thpt(V_f, p_f)$) where $p_f = 1$ and $V_f = 1$. Once search space and initial values are set, a brute force search (search space is small) is employed to find p_f and V_f such that $Thpt(V_f, p_f)$ is maximum and resource consumption is within FPGA resource limit.

The algorithm for spatially blocked design also employs a similar strategy but it always keeps the on-chip memory requirement within the resource limit by scaling down the size of spatial blocks. Hence, on-chip memory resources will no longer be determining factors for maximum design parameters. As spatially blocked design does overlap computation, throughput ($Thpt(V_f, p_f, Tile)$) will depend on spatial block size, $Tile$. Similar to the baseline approach, it explores all the design space and determines the design parameters which give maximum throughput while required resources are within FPGA resource limit. in addition to design parameters p_f and V_f, corresponding spatial block size, $Tile$ will also determined.

Algorithm 9: `Spatial blocking approach`

```
1: $r0 \leftarrow \sum_{i=0}^{\#kernels} RNode(i,1)$
2: $k0 \leftarrow max(r0.DSP/F.DSP, r0.LUT/F.LUT, r0.Reg/F.Reg)$
3: $p\_max \leftarrow 1/k0$
4: $V_{maxRes} \leftarrow 1/k0$
5: $V_{maxBW} \leftarrow \frac{total\_banwdidth}{2 \times f \times sizeof(DType) \times no\_of\_global\_nodes}$
6: $V_{max} \leftarrow min(V_{maxRes}, V_{maxBW})$
7: $V_f \leftarrow 1$
8: $p_f \leftarrow 1$
9: $Tile_f \leftarrow calTile(0)$
10: $T_{max} \leftarrow Thpt(1,1)$
11: for $p = 1, 2, ..., p\_max$ do
12:     for $V = 1, 2, ..., v\_max$ do
13:         $r \leftarrow \sum_{i=0}^{\#kernels} RNode(i,V)$
14:         $k \leftarrow max(r.DSP/F.DSP, r.LUT/F.LUT, r.Reg/F.Reg)$
15:         $Tile \leftarrow calTile(p)$
16:         if $K > 1$ then
17:             continue
18:         end if
19:         $T \leftarrow ThptT(V, p, Tile)$
20:         if $T > T_{max}$ then
21:             $T_{max} \leftarrow T$
22:             $V_f \leftarrow V$
23:             $p_F \leftarrow p$
24:             $Tile_f \leftarrow Tile$
25:         end if
26:     end for
27: end for
```

6.4 Discussion and Concluding Remarks

This Chapter proposed a methodology for transforming structured mesh-based numerical applications specified using a DSL to FPGA target language, specifically C++ for Vivado. This approach is motivated by the profitability of FPGAs for a subset of such applications. Unlike existing state-of-the-art automatic translators, our methodology focuses utilizing proven DSL to specify this class of applications and on transformation techniques and optimizations such as batching and tiling that are required for realistic applications. Furthermore, we provide several FPGA-specific optimizations to improve the quality of results, such as device area and clock frequency on FPGAs. The transformation steps are designed in a way that an automatic translator can be built using modern compiler frameworks. However, it should be noted that although the provided methodology can be used to translate a wider range of applications, only a subset of applications will yield good performance on FPGAs. One major limitation of the current generation of FPGAs is their larger reconfiguration time, which requires all nodes in the dataflow graph to fit into the FPGA. If the resource consumption exceeds the available resources on the FPGA, it is impossible for the nodes to fit, and even if they marginally fit, the operating frequency will be low due to possible routing congestion.

Chapter 7

Conclusions and Future Work

The utilization of FPGAs has gained significant traction within the HPC community, primarily due to their exceptional processing speeds, low latency, and energy efficiency when deployed in specific domains. Nevertheless, both industry and research are constrained in their ability to utilize FPGAs for HPC applications due to the challenging nature of programming these devices. Despite the introduction of HLS tools and High-Level APIs by FPGA vendors, developing a high-performing FPGA-based implementation for an application remains a formidable task. The difficulty arises primarily from the requirement of FPGA-specific transformations that demand a high level of proficiency in digital system design and data flow programming models.

In this thesis, we have attempted to bridge the gap in getting better performance and ease of programming FPGAs through a domain-specific approach. The key characteristics of application class, their computation and communication patterns or motifs is leveraged in this approach to explore the design space on the target accelerator device. We applied such analysis to the widely used domain of structured mesh-based numerical algorithms, specifically for (1) explicit stencil solvers characterised by looping through rectangular mesh and accessing mesh elements using a stencil and (2) tridiagonal systems-based implicit solvers, which solve linear systems along each dimension of mesh.

Several prior studies have investigated and examined the use of FPGAs for structured mesh-based explicit applications. However, these analyses have been primarily limited to evaluating the performance of individual applications on FPGAs, without developing a comprehensive methodology. Additionally, key optimizations required for realistic workloads, such as batched and tiled computation for implicit applications, have not been explored for this application class on FPGAs. Furthermore, previous works have not clearly demonstrated the codification of FPGA-specific optimizations using high-level languages. As a result, this study aimed to consolidate the optimization techniques of prior research, alongside new optimization strategies, to enable the optimal implementation of realistic applications on FPGAs.

The current state of the art in accelerating implicit numerical algorithms is primarily focused on implementing single tridiagonal system solvers on FPGA, rather than analyzing and implementing full applications. This thesis proposed a systematic approach for evaluating the performance of different tridiagonal solver algorithms for implicit applications through models. Based on the characteristics of the applications, several optimization techniques are presented in this thesis to save memory bandwidth and improve performance on modern FPGAs. Using these techniques, a new multi-dimensional tridiagonal solver library is developed, along with data path transformation routines for data path design. This new library achieves over one order of magnitude improvement in throughput compared to the state of the art tridiagonal solvers for larger batches of systems.

7.1 Contributions and Conclusions

In order to address the above key gaps in the research, Chapter 3 investigated and presented methodology to accelerate the structured mesh-based explicit numerical application on FPGAs, Chapter 4 explored implicit application on FPGAs and chapter 5 explored the acceleration of both implicit and explicit application on FPGAs using SYCL programming model.

Chapter 3 and 5 presented the first main contribution involving the development of a unified workflow and implementation template for implementing structured mesh-based numerical explicit and implicit numerical algorithms on FPGAs. The batched execution of stencil solvers on independent meshes presented in this thesis is a novel approach. It demonstrates that FPGAs can not only provide low latency for processing a single mesh but can also be utilized to obtain higher throughput for solving set of meshes.

The second main contribution (Chapter 4) comes from the development of a high-throughput multi-dimensional tridiagonal system solver library with a design space exploration technique for implementing implicit applications on FPGAs. A design and optimization strategy is proposed based on the size and dimensionality of the mesh, along with dataflow path optimizations. This library achieves over one order of magnitude improvement in speed compared to the state-of-the-art Xilinx tridiagonal system solver library.

The third main contribution (also in Chapters 3 in 5) pertains to the development of analytical models that aid in predicting performance and facilitating design exploration. The predictive performance model, along with models for key resource consumption, provides a systematic approach to determining optimal design parameters, selecting an optimization strategy, and evaluating the profitability of FPGA implementations. The models presented in this thesis demonstrate over 85% accuracy in predicting runtime compared to actual results.

The above contributions are supported by benchmarking realistic applications on modern FPGAs using developed methodology with performance of same application on HPC grade GPUs. Targetting Xilinx and Intel FPGAs, we present design and optimization of three representative explicit stencil solver based applications and two implicit tridiagonal solver based application, comparing different optimizations and performance trade-offs.

These representative applications include 2D and 3D solvers using both FP32 and FP64 arithmetic, operating on both scalar and vector mesh-points. Detailed performance analysis in terms of runtime, power and bandwidth is presented comparing implementation on FPGAs with highly optimized implementation on HPC grade Nvidia V100 GPU. The results of the benchmarking show that FPGAs provide competitive performance compared to the same generation GPUs while consuming over 30% less energy in GPU-based computation.

A final contribution (Chapter 6) of this research pertains to the proposed automatable workflow based on design and optimization techniques developed in this work for FPGA implementation of this class of applications. Utilizing the DSL of popular OPS framework, this work presents a generalized high level technique based on design templates/skeletons to transform it to FPGA target implementation. The transformation technique is devised such that it can be automatable using modern compiler techniques.

7.2 Future Work

There are several potential areas for future research based on the work in this thesis. They can be broadly categorized into two orthogonal areas. Firstly, there is a need for support for larger applications with memory and FPGA resource requirements that exceed the capabilities of a single FPGA device. Secondly, there is scope for the automatic translation of structured mesh-based numerical applications specified using a DSL such as OPS. The skeleton and steps presented in Chapter 6 can be utilized alongside modern compiler tools to generate FPGA target implementations automatically.

7.2.1 Support for Larger Meshes

This thesis presented a workflow for executing structured mesh-based applications on FPGA boards with memory requirements that fall within the device's storage capacity. Additionally, the resource demands of chained kernels are constrained to within the FPGA's resource limit. Scientific applications, such as Computational Fluid Dynamics (CFD), often necessitate large data structures, leading to implementation on multi-node accelerator devices to ensure sufficient aggregate memory. However, these implementations commonly face communication overhead bottlenecks during the exchange of halo regions between nodes, with users lacking direct control over node communication.

In FPGA implementations, larger memory requirements can be addressed in two ways. Firstly, if the host has sufficient memory to accommodate the corresponding data structure, the entire structure can reside on the host, and a tile block can be moved to and processed on the device before being returned to the host. This approach is similar to host-based spatial blocking, but the implementation's performance is constrained by the bandwidth between the host and device. While modern devices like Intel Agilex 7 devices with PCIe 5.0 x16 can support around 128GB/s throughput, this speed is much lower than that of HBM memory throughput. The time required for data movement can be minimized by overlapping computation and communication and utilizing the tile blocks moved to the device for multiple steps. However, the tile blocks sent to the device are likely to be large enough to necessitate another spatially blocked implementation within the FPGA, creating a multi-level tiling implementation capable of supporting larger meshes that exceed the accelerator device's memory capacity.

Alternatively, a larger data structure can be divided into multiple chunks and mapped to many FPGA accelerator devices. Each device can then compute on its designated chunk, often exchanging boundary/halo data with other nodes. This method provides better performance than the first approach since multiple chunks can be processed in parallel. To facilitate data exchange, a dedicated communication link can be established between FPGA devices via a network interface. These links can help overcome communication bottlenecks that arise in traditional multi-node GPU and CPU accelerator implementations.

7.2.2 Support for Larger Number of Kernels

Instructions-based accelerators are capable of loading kernels within microseconds, and the overhead of loading kernels in sequence is insignificant if the runtime of each kernel is sufficiently long. Therefore, executing a set of kernels in an iterative loop does not incur a significant overhead for kernel loading. However, on FPGAs, loading a set of kernels requires partial reconfiguration of the device, which can take several seconds, often longer than the runtime of the kernel call in the iterative loop. As a result, the set of kernels in the iterative loop must be chained together and loaded in a single partial reconfiguration, which imposes area/resource constraints on each kernel in the iterative loop. In order to support a larger number of kernels, researchers have attempted to implement structured mesh-based numerical algorithms, particularly explicit stencil solvers [15], on multiple FPGAs. The aggregate resources available in multiple FPGAs enable longer kernel pipelines. Communication between devices is limited by network and PCIe communication speeds, but these links have achieved significant improvements in bandwidth for data movement between kernels. However, previous works on multi-FPGA implementation have been limited to baseline designs, and realistic application optimizations such as batching and spatial blocking have not been explored in multi-FPGA implementations. Such an implementation would broaden the scope of FPGAs for this class of applications and provide insights into the profitability of using FPGAs for larger structured mesh-based numerical applications.

7.2.3 DSL based Automatic Translator

The first part of the thesis focuses on developing an optimized workflow for implementing structured mesh-based numerical applications on FPGAs. Chapter 6 utilizes the techniques presented in Chapters 3–5 to propose a methodology for codifying FPGA kernels using skeletons for OPS applications. While this approach simplifies the development of FPGA target kernels, significant effort is still required to explore the design space and find the optimal design parameters to codify the kernels.

The implementation of FPGA kernels requires lower-level customizations and multiple FPGA-specific transformations, resulting in a significantly larger number of lines in the target language implementation. This not only requires a longer implementation time but also leads to longer verification and debugging times due to possible user implementation mistakes. Additionally, FPGA HLS tools are continuously evolving, resulting in different `#pragma` usage across various HLS compilers, and new FPGA devices with different hardware specifications are being introduced to the market, making it time-consuming to explore the design space for each new hardware device and implement the application for the target device.

To address a similar challenge, recent research works have focused on utilizing domain-specific frameworks, such as OPS, to automatically generate target implementations for parallel architectures such as GPUs and CPUs. OPS is capable of applying optimizations specific to the target architecture, resulting in implementations that achieve performance similar to that of hand-written code. Additionally, applications written using these domain-specific languages (DSLs) do not need to be updated to support new devices and target compilers, only the translator needs to be updated. This is potentially a pathway for future-proofing the applications.

The authors of [4] developed a skeleton-based domain-specific language (DSL) translator for OPS DSL that generates target kernels using the LLVM framework. The ideas in this approach can be extended to generate FPGA target kernels using the techniques presented in Chapter 6. The LLVM framework offers a rich set of functions to analyze front-end specifications, such as syntax errors, data types, and semantic analysis. Since OPS is a C++-based embedded DSL, the benefits of the LLVM framework can be leveraged to create a production-quality FPGA kernel translator. Moreover, since OPS uses the same host program for all target platforms, the same host program can be used for FPGA targets as well, provided that library functions for data movement between kernel and device are implemented.

****

Modern FPGA accelerator devices not only provide energy-efficient and low-latency processing but also provides high throughput for certain classes of applications. The introduction of high-level synthesis tools significantly improved the FPGA application development productivity. Along with continuous improvement on HLS tools and the introduction of new powerful HPC grade FPGAs, high-level abstraction using domain-specific languages could enable domain scientists to take full advantage of the benefits provided by FPGAs. We look forward to the wider use of FPGAs for HPC workloads and expect this work will contribute to demonstrating an accessible path for their extended utility.

Bibliography

[1] Addressing Memory-Bandwidth and Compute-Intensive Challenges with Intel® Agilex™ 7 FPGAs M-Series. Addressing Memory-Bandwidth and Compute-Intensive Challenges with Intel® Agilex™ 7 FPGAs M-Series, 2022. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/memory-bandwidth-and-compute-intensive-with-agilex-m-series-white-paper.pdf.

[2] J. Babb, R. Tessier, and A. Agarwal. Virtual wires: overcoming pin limitations in FPGA-based logic emulators. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, 1993. doi: 10.1109/FPGA.1993.279469.

[3] G. D. Balogh, T. Flynn, S. Laizet, G. R. Mudalige, and I. Z. Reguly. Scalable Many-core Algorithms for Tridiagonal Solvers. *Journal of Computing in Science and Engineering*, 2021. (In Press).

[4] G.D. Balogh, G.R. Mudalige, I.Z. Reguly, S.F. Antao, and C. Bertolli. OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 59–70, 2018. doi: 10.1109/LLVM-HPC.2018.8639205.

[5] Tobias Becker, Oskar Mencer, Stephen Weston, and Georgi Gaydadjiev. *Maxeler Data-Flow in Computational Finance*, pages 243–266. Springer International Publishing, Cham, 2015. ISBN 978-3-319-15407-7. doi: 10.1007/978-3-319-15407-7_11. URL https://doi.org/10.1007/978-3-319-15407-7_11.

[6] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, 2019.

[7] Neil W. Bergmann, Sunil K. Shukla, and Jürgen Becker. QUKU: A Dual-Layer Reconfigurable Architecture. *ACM Trans. Embed. Comput. Syst.*, 12(1s), mar 2013. ISSN 1539-9087. doi: 10.1145/2435227.2435259. URL https://doi.org/10.1145/2435227.2435259.

[8] Alexander Brant and Guy G.F. Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 93–96, 2012. doi: 10.1109/FCCM.2012.25.

[9] P. Nithiarasu C. Zienkiewicz, R. L. Taylor and J. Zhu. *"The finite element method, volume 3"*. McGraw-Hill, London, 1977.

[10] 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)*, pages 1–8, 2018.

[11] Robert Clayton and Björn Engquist. Absorbing boundary conditions for acoustic and elastic wave equations. *Bulletin of the Seismological Society of America*, 67(6): 1529–1540, 12 1977. ISSN 0037-1106.

[12] J. Cong and J. Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[13] David Bruce Cousins, Kurt Rohloff, and Daniel Sumorok. Designing an FPGA-accelerated homomorphic encryption co-processor. *IEEE Transactions on Emerging Topics in Computing*, 5(2):193–206, 2016.

[14] cuSPARSE API Reference. cuSPARSE API Reference, Oct 2021. [https://docs.nvidia.com/cuda/cusparse/index.html].

[15] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 315–326. IEEE Press, 2021. ISBN 9781728186139. doi: 10.1109/CGO51591.2021.9370315. URL https://doi.org/10.1109/CGO51591.2021.9370315.

[16] Discussions with the Numerical Algorithms Group, UK. Discussions with the Numerical Algorithms Group, UK., 2019.

[17] K. Dohi, K. Fukumoto, Y. Shibata, and K. Oguri. Performance modeling and optimization of 3-D stencil computation on a stream-based FPGA accelerator. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2013.

[18] Jim Douglas and James E Gunn. A General Formulation of Alternating Direction Methods. *Numèrische mathèmatik*, 6(1):428–453, 1964.

[19] DSP HDL Toolbox. DSP HDL Toolbox, Design digital signal processing applications for FPGAs, ASICs, and SoCs, 2023. https://uk.mathworks.com/products/dsp-hdl.html.

[20] Robert Eymard, Thierry Gallouët, and Raphaèle Herbin. Finite volume methods. *Handbook of numerical analysis*, 7:713–1018, 2000.

[21] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized FPGA accelerators for efficient cloud computing. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435, 2015.

[22] FPGA Optimization Guide for Intel® oneAPI Toolkits. FPGA Optimization Guide for Intel® oneAPI Toolkits, 2023. https://www.intel.com/content/dam/develop/external/us/en/documents/oneapi-dpcpp-fpga-optimization-guide.pdf.

[23] Haohuan Fu and Robert G. Clapp. Eliminating the Memory Bottleneck: An FPGA-Based Solution for 3d Reverse Time Migration. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 65–74, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305549. doi: 10.1145/1950413.1950429. URL https://doi.org/10.1145/1950413.1950429.

[24] Walter Gander and Gene H Golub. Cyclic Reduction—History and Applications. *Scientific computing (Hong Kong, 1997)*, 7385, 1997.

[25] Getting Started with Vitis HLS. Getting Started with Vitis HLS, 2023. https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/Getting-Started-with-Vitis-HLS.

[26] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. STELLA: a domain-specific tool for structured grid methods in weather and climate models. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015. doi: 10.1145/2807591.2807627.

[27] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[28] Mike Hutton. Understanding How the New Intel® HyperFlex™ FPGA Architecture Enables Next-Generation High-Performance Systems, 2022. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01231-understanding-how-hyperflex-architecture-enables-high-performance-systems.pdf/.

[29] Intel® FPGA Programmable Acceleration Card D5005. Intel® FPGA Programmable Acceleration Card D5005, 2022. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/pac/d5005.html.

[30] Christian T. Jacobs, Satya P. Jammy, and Neil D. Sandham. OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *Journal of Computational Science*, 18:12–23, 2017. ISSN 1877-7503. doi: https://doi.org/10.1016/j.jocs.2016.11.001. URL https://www.sciencedirect.com/science/article/pii/S187775031630299X.

[31] Abhishek Kumar Jain, Suhaib A. Fahmy, and Douglas L. Maskell. Efficient Overlay Architecture Based on DSP Blocks. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28, 2015. doi: 10.1109/FCCM.2015.15.

[32] Q. Jia and H. Zhou. Tuning Stencil codes in OpenCL for FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 249–256, 2016.

[33] Kamalavasan Kamalakkannan, Gihan R. Mudalige, István Z. Reguly, and Suhaib A. Fahmy. High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1087–1096, 2021. doi: 10.1109/IPDPS49936.2021.00117.

[34] Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. FPGA Acceleration of Structured-Mesh-Based Explicit and Implicit Numerical Solvers Using SYCL. In *International Workshop on OpenCL*, IWOCL'22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396585. doi: 10.1145/3529538.3530007. URL https://doi.org/10.1145/3529538.3530007.

[35] Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. High Throughput Multidimensional Tridiagonal System Solvers on FPGAs. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392815. doi: 10.1145/3524059.3532371. URL https://doi.org/10.1145/3524059.3532371.

[36] Kamalavasan Kamalakkannan, Gihan Mudalige, István Zoltán Reguly, and Suhaib Fahmy. FPGA tridiagonal solver library, March 2023. URL https://doi.org/10.5281/zenodo.7750458.

[37] Kamalavasan Kamalakkannan, Beniel Thileepan, Gihan Mudalige, István Zoltán Reguly, and Suhaib Fahmy. Benchmarking - Xilinx Alveo U280 vs Nvidia V100 for Stencil applications, March 2023. URL https://doi.org/10.5281/zenodo.7750461.

[38] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. High Bandwidth Memory on FPGAs: A Data Analytics Perspective.

In *International Conference on Field-Programmable Logic and Applications (FPL)*, 2020.

[39] T. Kenter, J. Förstner, and C. Plessl. Flexible FPGA design for FDTD using OpenCL. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, 2017.

[40] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 242–251, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361378. doi: 10.1145/3289602.3293910. URL https://doi.org/10.1145/3289602.3293910.

[41] Michael Lange, Navjot Kukreja, Mathias Louboutin, Fabio Luporini, Felippe Vieira, Vincenzo Pandolfo, Paulius Velesko, Paulius Kazakas, and Gerard Gorman. Devito: Towards a generic finite difference dsl using symbolic python. *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 67–75, 2016.

[42] Endre Laszlo, Mike Giles, and Jeremy Appleyard. Manycore Algorithms for Batch Scalar and Block Tridiagonal Solvers. *ACM Transactions on Mathematical Software (TOMS)*, 42(4):1–36, 2016.

[43] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, 2007. doi: 10.1137/1.9780898717839. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898717839.

[44] H. Levy and F. Lessman. *"Finite difference equations"*. Courier Corporation, 1992.

[45] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.*, 46(1), apr 2020. ISSN 0098-3500. doi: 10.1145/3374916. URL https://doi.org/10.1145/3374916.

[46] Roman Lysecky, Kris Miller, Frank Vahid, and Kees Vissers. Firm-Core Virtual FPGA for Just-in-Time FPGA Compilation (Abstract Only). In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, FPGA '05, page 271, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930299. doi: 10.1145/1046192.1046247. URL https://doi.org/10.1145/1046192.1046247.

[47] Endre László, Zoltán Nagy, Michael B. Giles, István Reguly, Jeremy Appleyard, and Peter Szolgay. Analysis of Parallel Processor Architectures for the Solution of the Black-Scholes PDE. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1977–1980, 2015. doi: 10.1109/ISCAS.2015.7169062.

[48] H. Macintosh, Jasmine Banks, and N. Kelson. Implementing and Evaluating an Heterogeneous, Scalable, Tridiagonal Linear System Solver with OpenCL to Target FPGAs, GPUs, and CPUs. *Int. J. Reconfigurable Comput.*, 2019:3679839:1–3679839:13, 2019.

[49] H. J. Macintosh, D. J. Warne, N. A. Kelson, J. E. Banks, and T. W. Farrell. Implementation of Parallel Tridiagonal Solvers for a Heterogeneous Computing Environment. In Jason Sharples and Judith Bunder, editors, *Proceedings of the 17th Biennial Computational Techniques and Applications Conference, CTAC-2014*, volume 56 of *ANZIAM J.*, pages C446–C462, Feb 2016. URL `http://journal.austms.org.au/ojs/index.php/ANZIAMJ/article/view/9371`.

[50] GR Mudalige, MB Giles, I Reguly, C Bertolli, and PHJ Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. *2012 Innovative Parallel Computing, InPar 2012*, 2012. doi: 10.1109/InPar.2012.6339594. URL `http://dx.doi.org/10.1109/InPar.2012.6339594`.

[51] G.R. Mudalige, I.Z. Reguly, S.P. Jammy, C.T. Jacobs, M.B. Giles, and N.D. Sandham. Large-scale performance of a DSL-based multi-block structured-mesh application for Direct Numerical Simulation. *Journal of Parallel and Distributed Computing*, 131:130 – 146, 2019. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2019.04.019.

[52] 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)*, pages 1–8, 2016.

[53] NVIDIA V100 Data Sheet. NVIDIA V100 Data Sheet, Jan 2020. `https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf`.

[54] Filipe Oliveira, C. Silva Santos, F. A. Castro, and José C. Alves. A Custom Processor for a TDMA Solver in a CFD Application. In Roger Woods, Katherine Compton, Christos Bouganis, and Pedro C. Diniz, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, pages 63–74, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78610-8.

[55] OpenCL Specification. OpenCL Specification, 2023. https://registry.khronos.org/OpenCL/specs/2.2/html/OpenCL_API.html.

[56] OPS for Many-Core Platforms. OPS for Many-Core Platforms, 2014. https://github.com/OP-DSL/OPS.

[57] OPS Manual. OPS Manual, 2023. https://ops-dsl.readthedocs.io/en/latest/.

[58] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid CPU-FPGA databases. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218, 2017.

[59] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989. doi: 10.1109/43.31522.

[60] Eric Polizzi and Ahmed H. Sameh. A Parallel Hybrid Banded System Solver: the SPIKE Algorithm. *Parallel Computing*, 32(2):177–194, 2006. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2005.07.005. Parallel Matrix Algorithms and Applications (PMAA'04).

[61] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Softw.*, 43(3), dec 2016. ISSN 0098-3500. doi: 10.1145/2998441. URL https://doi.org/10.1145/2998441.

[62] Prashant Rawat, Martin Kong, Tom Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. SDSLc: A Multi-Target Domain-Specific Compiler for Stencil Computations. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340168. doi: 10.1145/2830018.2830025. URL https://doi.org/10.1145/2830018.2830025.

[63] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):873–886, April 2018. ISSN 1045-9219. doi: 10.1109/TPDS.2017.2778161.

[64] Istvan Z. Reguly, Branden Moore, Tim Schmielau, Jacques du Toit, and Gihan R. Mudalige. Batch solution of small PDEs with the OPS DSL. In Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode, editors, *High Performance Computing*, pages 124–141, Cham, 2019. Springer International Publishing. ISBN 978-3-030-34356-9.

[65] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *"Unified Shared Memory"*, pages 149–171. Apress, Berkeley, CA, 2021. ISBN 978-1-4842-5574-2. doi: 10.1007/978-1-4842-5574-2_6. URL https://doi.org/10.1007/978-1-4842-5574-2_6.

[66] Bajaj Ronak and Suhaib A Fahmy. Mapping for maximum performance on FPGA DSP blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(4):573–585, 2015.

[67] 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*, pages 234–241, 2011.

[68] 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*, 25(3):695–705, 2014.

[69] M. Schmidt, M. Reichenbach, and D. Fey. A Generic VHDL Template for 2D Stencil Code Applications on FPGAs. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 180–187, 2012.

[70] M. Shafiq, M. Pericàs, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguadé. Exploiting memory customization in FPGA for 3D stencil computations. In *2009 International Conference on Field-Programmable Technology*, pages 38–45, 2009.

[71] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gomez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 9–7, 2020.

[72] Hayden Kwok-Hay "So and Cheng" Liu. *"FPGA Overlays"*, pages 285–305. Springer International Publishing, Cham, 2016. ISBN 978-3-319-26408-0. doi: 10.1007/978-3-319-26408-0_16. URL https://doi.org/10.1007/978-3-319-26408-0_16.

[73] SYCL 2020 Specification. SYCL 2020 Specification, 2023. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html.

[74] G. Tataru and T. Fisher. Stochastic Local Volatility. *Quantitative Development Group, Bloomberg Version 1*, Feb 5 2010.

[75] J. W. Thomas. *"Numerical Partial Differential Equations: Finite Difference Methods"*. Springer New York, NY, 1995.

[76] Llewellyn Thomas. Elliptic Problems in Linear Differential Equations Over a Network: Watson Scientific Computing Laboratory. *Columbia Univ., NY*, 1949.

[77] Tridsolver Library. Tridsolver Library, July 2020. https://github.com/OP-DSL/tridsolver.

[78] Pedro Valero-Lara, Ivan Martínez-Pérez, Raül Sirvent, Xavier Martorell, and Antonio J. Peña. NVIDIA GPUs Scalability to Solve Multiple (Batch) Tridiagonal Systems Implementation of cuThomasBatch. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 243–253, Cham, 2018. Springer International Publishing. ISBN 978-3-319-78024-5. doi: 10.1007/978-3-319-78024-5_22.

[79] P. Vincent, F. Witherden, B. Vermeire, J. S. Park, and A. Iyer. Towards Green Aviation with Python at Petascale. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2016. doi: 10.1109/SC.2016.1.

[80] Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance. Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance, 2022. https://docs.xilinx.com/v/u/en-US/wp485-hbm.

[81] Vision HDL Toolbox. Vision HDL Toolbox, Design image processing, video, and computer vision systems for FPGAs and ASICs, 2023. https://uk.mathworks.com/products/vision-hdl.html.

[82] Vitis Quantitative Finance Library V.2020.2. Vitis Quantitative Finance Library V.2020.2, 2020. https://xilinx.github.io/Vitis_Libraries/quantitative_finance/2020.2/.

[83] H. M. Waidyasooriya and M. Hariyama. Multi-FPGA Accelerator Architecture for Stencil Computation Exploiting Spacial and Temporal Scalability. *IEEE Access*, 7: 53188–53201, 2019.

[84] 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*, 28(5):1390–1402, 2017.

[85] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. DLAU: A scalable deep learning accelerator unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[86] Xinliang Wang, Yangtong Xu, and Wei Xue. A Hierarchical Tridiagonal System Solver for Heterogenous Supercomputers. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 69–76, 2014. doi: 10.1109/ScalA.2014.12.

[87] David Warne, Neil Kelson, and Ross Hayward. Solving Tri-diagonal Linear Systems Using Field Programmable Gate Arrays. In Y Gu and S Saha, editors, *Proceedings of the 4th International Conference on Computational Methods*, pages 1–8. Queensland University of Technology, Australia, 2012. URL https://eprints.qut.edu.au/54894/.

[88] David J. Warne, Neil A. Kelson, and Ross F. Hayward. Comparison of High Level FPGA Hardware Design for Solving Tri-diagonal Linear Systems. *Procedia Computer Science*, 29:95–101, 2014. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2014.05.009. URL https://www.sciencedirect.com/science/article/pii/S1877050914001860. 2014 International Conference on Computational Science.

[89] Wireless HDL Toolbox. Wireless HDL Toolbox, Design and implement 5G and LTE communications subsystems for FPGAs, ASICs, and SoCs, 2023. https://uk.mathworks.com/products/wireless-hdl.html.

[90] Xilinx - Large FPGA methodology guide. Xilinx - Large FPGA methodology guide, 2012. https://www.xilinx.com/htmldocs/xilinx14_4/ug872_largefpga.pdf.

[91] *Alveo U280 Data Center Accelerator Card Data Sheet*. Xilinx Inc., May 2020. v1.3.

[92] YASK–Yet Another Stencil Kit. YASK–Yet Another Stencil Kit, 2023. https://github.com/intel/yask.

[93] Wei Zhang, Vaughn Betz, and Jonathan Rose. Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver. *ACM Trans. Reconfigurable Technol. Syst.*, 5(1), Mar 2012. ISSN 1936-7406. doi: 10.1145/2133352.2133358. URL https://doi.org/10.1145/2133352.2133358.

[94] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. *"AutoPilot: A Platform-Based ESL Synthesis System"*, pages 99–112. Springer Netherlands, Dordrecht, 2008. ISBN 978-1-4020-8588-8. doi: 10.1007/978-1-4020-8588-8_6. URL https://doi.org/10.1007/978-1-4020-8588-8_6.

[95] 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)*, pages 123–130, 2018.

[96] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 153–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356145. doi: 10.1145/3174243.3174248. URL https://doi.org/10.1145/3174243.3174248.

Appendix A

2D Heat diffusion using FDM

In Chapter 2, the 2D-Poisson equation is solved through three different numerical schemes and an analysis of their suitability for parallel architectures based on compute and communication patterns was presented. This appendix section illustrates the numerical stability of those schemes through a 2D heat equation that has a similarity with the 2D heat application we used for benchmarking FPGAs and GPUs in Chapter 4-5.

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial^2 x} + \frac{\partial^2 T}{\partial^2 y} \right) \tag{A.1}$$

Above abstract 2D heat equation can be attempted to be solved using the finite difference method. We discretize the 2D space x, y into i, j and time t into k, as such the temperature T using discretized space would be $T^k_{i,j}$. We note the following boundary condition and initial values for this problem:

- Initial temperature (T) at 2D space ($T^0_{i,j}$) is given
- Temperature (T) along the boundary is known at any time step k

A.1 FTCS - Explicit Numerical Scheme

In Equation A.1, the forward difference can be used to compute $\frac{\partial T}{\partial t}$ with the resolution of Δt and second order central difference can be used to rewrite $\frac{\partial^2 T}{\partial^2 x}, \frac{\partial^2 T}{\partial^2 y}$ with the resolution of h for both x and y. It is called the Forward Time Centered Space (FTCS) method and the resulting equation will be as follows:

$$\frac{T^{k+1}_{i,j} - T^k_{i,j}}{\Delta t} = \alpha \left(\frac{T^k_{i-1,j} - 2T^k_{i,j} + T^k_{i+1,j}}{h^2} + \frac{T^k_{i,j-1} - 2T^k_{i,j} + T^k_{i,j+1}}{h^2} \right) \tag{A.2}$$

Above equation can be re-written as:

$$T^{k+1}_{i,j} = \left(1 - \frac{4\Delta t \alpha}{h^2} \right) T^k_{i,j} + \Delta t \alpha \left(T^k_{i-1,j} + T^k_{i+1,j} + T^k_{i,j-1} + T^k_{i,j+1} \right) \tag{A.3}$$

Values in time step $k+1$ can be obtained using a 5-point stencil that accesses the values in time step k. However, it can be shown through analysis of the amplification factor that the above equation is stable only if:

$$\left(1 - \frac{4\Delta t\alpha}{h^2}\right) > 0 \tag{A.4}$$

It can be re-written as

$$\Delta t < \frac{h^2}{4\alpha} \tag{A.5}$$

The following can be inferred from the above condition:

- This explicit scheme is conditionally stable
- Smaller resolution in space would require a much smaller resolution in the time domain, hence it would require a huge number of iterations

A.2 BTCS Numerical Scheme

In Equation A.2, FTCS scheme used forward difference of $\frac{\partial T}{\partial t}$. Instead, the backward difference can be used for $\frac{\partial T}{\partial t}$ and the resulting equation will be as follows.

$$\frac{T^k_{i,j} - T^{k-1}_{i,j}}{\Delta t} = \alpha\left(\frac{T^k_{i-1,j} - 2T^k_{i,j} + T^k_{i+1,j}}{h^2} + \frac{T^k_{i,j-1} - 2T^k_{i,j} + T^k_{i,j+1}}{h^2}\right) \tag{A.6}$$

This scheme is called Backward Time Centered Space (BTCS) method and the above equation can be arranged as follows:

$$aT^k_{i,j} - bT^k_{i-1,j} - bT^k_{i+1,j} - bT^k_{i,j-1} - bT^k_{i,j+1} = T^{k-1}_{i,j} \tag{A.7}$$

Here $a = 1 + \frac{4\alpha\Delta t}{h^2}$ and $b = \frac{\alpha\Delta t}{h^2}$. A system of equations, similar to one in Equation 2.7 can be formed by writing the above equation for all the i, j points along utilizing the initial and boundary values. This scheme is unconditionally stable. The drawback is solution method is hard to implement and parallelize.

A.3 ADI Scheme

In ADI scheme, $\frac{\partial^2 T}{\partial^2 x}$ is re-written using time step $k+1/2$ while $\frac{\partial T}{\partial t}$ and $\frac{\partial^2 T}{\partial^2 y}$ are re-written using time step k. This leads to the following equation:

$$\frac{T^{k+1/2}_{i,j} - T^k_{i,j}}{\Delta t/2} = \alpha\left(\frac{T^{k+1/2}_{i-1,j} - 2T^{k+1/2}_{i,j} + T^{k+1/2}_{i+1,j}}{h^2} + \frac{T^k_{i,j-1} - 2T^k_{i,j} + T^k_{i,j+1}}{h^2}\right) \tag{A.8}$$

It can be re-written as follows:

$$aT_{i,j}^{k+1/2} + bT_{i-1,j}^{k+1/2} + bT_{i+1,j}^{k+1/2} = cT_{i,j}^{k} + dT_{i,j-1}^{k} + dT_{i,j+1}^{k} \tag{A.9}$$

Here, $a = 1 + \frac{\alpha \Delta t}{h^2}$, $b = -\frac{\alpha \Delta t}{2h^2}$, $c = 1 - \frac{\alpha \Delta t}{h^2}$ and $d = \frac{\alpha \Delta t}{2h^2}$. A set of equations as A.9 can be formed as a tridiagonal system for each j. Similarly, another set of equations can be written using the time step $k+1$ for $\frac{\partial^2 T}{\partial^2 y}$ and using the time step $k+1/2$ for $\frac{\partial T}{\partial t}$ and $\frac{\partial^2 T}{\partial^2 x}$ as follows:

$$\frac{T_{i,j}^{k+1} - T_{i,j}^{k}}{\Delta t/2} = \alpha \left(\frac{T_{i-1,j}^{k+1/2} - 2T_{i,j}^{k+1/2} + T_{i+1,j}^{k+1/2}}{h^2} + \frac{T_{i,j-1}^{k+1} - 2T_{i,j}^{k+1} + T_{i,j+1}^{k+1}}{h^2} \right) \tag{A.10}$$

It can be re-written as follows:

$$aT_{i,j}^{k+1} + bT_{i,j-1}^{k+1} + bT_{i,j+1}^{k+1} = cT_{i,j}^{k+1/2} + dT_{i-1,j}^{k+1/2} + dT_{i+1,j}^{k+1/2} \tag{A.11}$$

This ADI scheme is unconditionally numerically stable for the 2D heat equation. However, it is only conditionally stable for the 3D heat equation.

Appendix B

Performance of Implicit Applications on U50

In Chapter 4, we benchmarked performance on Xilinx Alveo U280 for two representative applications with Nvidia V100. Here, instead of Xilinx Alveo U280, we benchmark the performance on Xilinx Alveo U50 with Nvidia V100. Xilinx Alveo U50 is a low-power (TDP 75W) accelerator device with two SLR regions and comes with HBM memory. Compared to U280, which consists of three SLR regions, we are limited by resources to scale the compute modules.

B.1 2D ADI Heat Diffusion Application

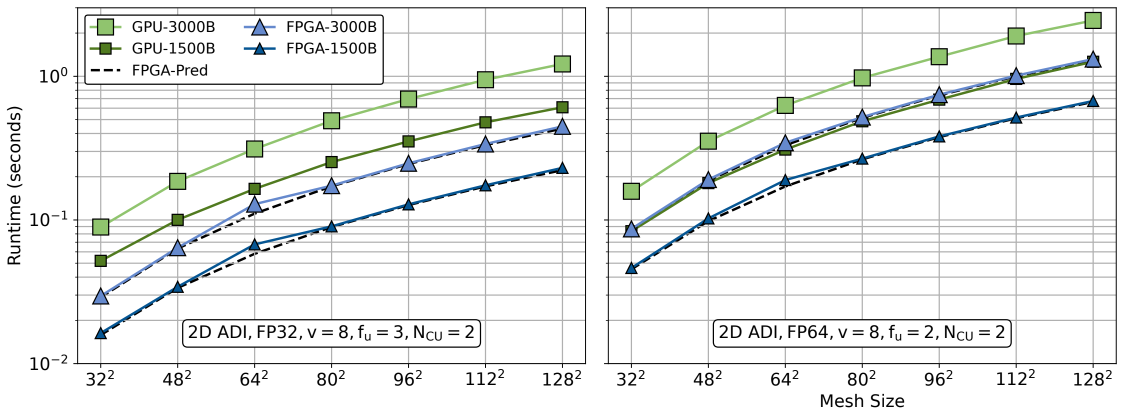


Figure B.1: 120 iterations.

B.2 3D ADI Heat Diffusion Application

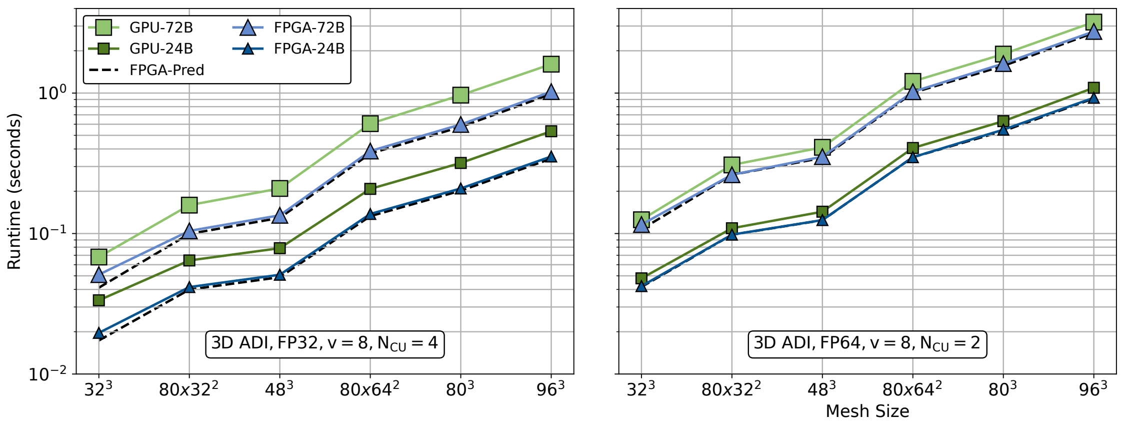


Figure B.2: 100 iterations.

B.3 2D ADI Heat Diffusion Application on Larger Meshes

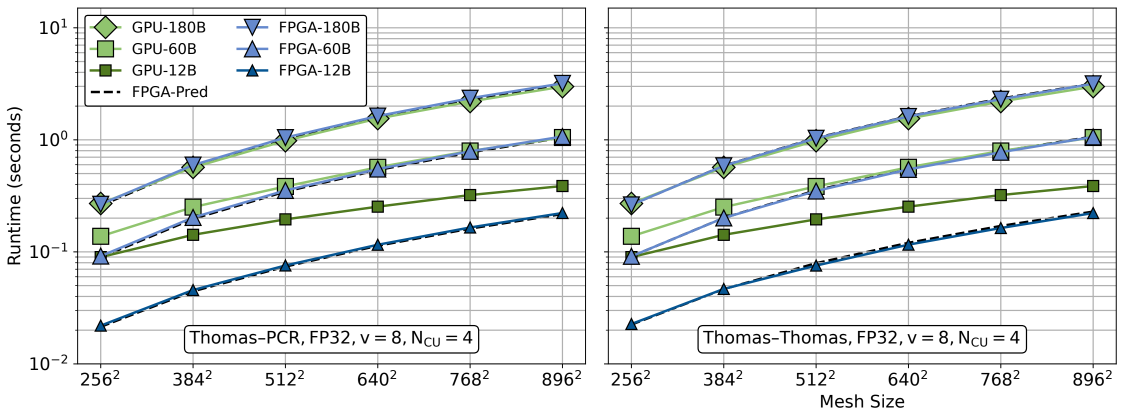


Figure B.3: 100 iterations.

B.4 SLV Application

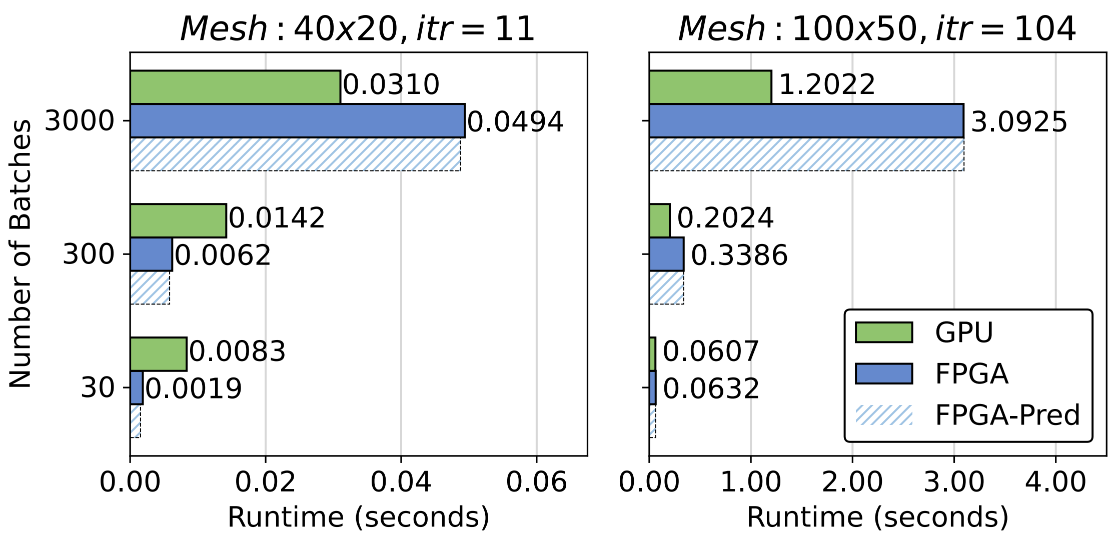


Figure B.4: SLV application

Appendix C

Runtimes of Benchmarked Applications

C.1 Chapter 3 Runtimes

Experimental system's specifications.

FPGA	Xilinx Alveo U280 [91]
DSP blocks	8490
BRAM / URAM	6.6MB (1487 blocks) / 34.5MB (960 blocks)
HBM	8GB, 460GB/s, 32 channels
DDR4	32GB, 38.4GB/s, in 2 banks (1 channel/bank)
Host	Intel Xeon Silver 4116 @2.10GHz (48 cores)
	256GB RAM, Ubuntu 18.04.3 LTS
Design SW	Vivado HLS, Vitis-2019.2
GPU	Nvidia Tesla V100 PCIe [91]
Global Mem.	16GB HBM2, 900GB/s
Host	Intel Xeon Gold 6252 @2.10GHz (48 cores)
	256GB RAM, Ubuntu 18.04.3 LTS
Compilers, OS	nvcc CUDA 9.1.85, Debian 9.11

Poisson-5pt-2D: Baseline (Figure 3.7, Runtimes in Seconds)

Mesh	200×100	200×200	300×150	300×300	400×200	400×400
GPU	0.506456	0.557750	0.434420	0.588450	0.577390	0.618230
FPGA	0.025010	0.035380	0.040370	0.063450	0.062740	0.104500
FPGA_Pred	0.023088	0.033488	0.041344	0.064144	0.064400	0.104400

Poisson-5pt-2D: Batched (Figure 3.8 (a), Runtimes in Seconds)

Mesh	200×100	200×200	300×150	300×300	400×200	400×400
GPU-100B	2.21160	3.84453	4.16839	7.60413	6.66065	12.76010
GPU-1000B	17.0044	33.3788	36.1999			
FPGA-100B	1.12066	2.16644	2.39700	4.68609	4.32076	8.49650
FPGA-1000B	11.0788	21.5203	23.8158			
Pred-100B	1.07328	2.11328	2.32864	4.60864	4.06400	8.06400
Pred-1000B	10.6205	21.0205	23.1222			

Poisson-5pt-2D: Tiled (Figure 3.8 (b), Runtimes in Seconds)

Mesh	512	1024	2048	4096	8000
GPU-15000^2	16.5599	16.5599	16.5599	16.5599	16.5599
GPU-20000^2	29.3727	29.3727	29.3727	29.3727	29.3727
FPGA-15000^2	16.3820	13.4231	12.4851	12.1019	11.9352
FPGA-20000^2	29.4516	23.9899	22.1574	21.8531	21.1726
Pred-15000^2	15.122	12.8960	12.0250	11.6378	11.4443
Pred-20000^2	26.8186	22.9551	21.4098	20.7659	20.3795

Jacobi-7pt-3D: Baseline (Figure 3.9, Runtimes in Seconds)

Mesh	50^3	100^3	150^3	200^3	250^3
GPU	0.324091	0.759514	1.60740	3.48595	6.03895
FPGA	0.143326	0.770618	2.25649	4.96895	9.27608
FPGA_Pred	0.136976	0.760439	2.23674	4.93175	9.21131

Jacobi-7pt-3D: Batched (Figure 3.10 (a), Runtimes in Seconds)

Mesh	50^3	100^3	150^3	200^3	250^3
GPU-10B	0.095170	0.498283	1.331192	3.210194	5.774286
GPU-50B	0.334188	2.303245	6.517967		
FPGA-10B	0.094334	0.613718	1.927346	4.403692	8.410590
FPGA-50B	0.448523	2.997540	9.489546		
Pred-10B	0.092839	0.608932	1.914211	4.374532	8.355746
Pred-50B	0.444579	2.977322	9.427707		

Jacobi-7pt-3D: Tiled (Figure 3.10 (b), Runtimes in Seconds)

Mesh	256	384	512	640	768
GPU-600^3	0.492143	0.492143	0.492143	0.492143	
GPU-$1800^2 \times 100$	0.798848	0.798848	0.798848	0.798848	0.7988487
FPGA-600^3	0.890816	0.731525	0.738536	0.710701	
FPGA-$1800^2 \times 100$	1.25812	1.25056	1.153999	1.140489	1.141086
Pred-600^3	0.778627	0.642515	0.642515	0.636175	
Pred-$1800^2 \times 100$	1.082243	1.071679	1.001397	0.998096	0.998096

RTM: Baseline (Figure 3.11 (a), Runtimes in Seconds)

Mesh	32^3	$32^2 \times 50$	$50^2 \times 16$	$50^2 \times 32$	50^3	$50^2 \times 200$	$50^2 \times 400$
GPU	0.138056	0.215500	0.175447	0.331792	0.457383	1.366103	2.585701
FPGA	0.331113	0.397877	0.565685	0.690383	0.830591	1.999264	3.557081
FPGA_Pred	0.323678	0.389885	0.556800	0.680533	0.819733	1.979733	3.526400

RTM: Batched (Figure 3.11 (b), Runtimes in Seconds)

Mesh	32^3	$32^2 \times 50$	$50^2 \times 16$	$50^2 \times 32$	50^3
GPU-20B	0.120959	0.222077	0.181801	0.415573	0.642531
GPU-40B	0.231278	0.427226	0.334811	0.804417	1.262631
FPGA-20B	0.318709	0.453378	0.416621	0.668576	0.952004
FPGA-40B	0.618231	0.887511	0.794693	1.298273	1.864900
Pred-20B	0.311908	0.444322	0.408320	0.655787	0.934187
Pred-40B	0.606161	0.870989	0.779520	1.274453	1.831253

C.2 Chapter 4 Runtimes

Experimental systems specifications.

FPGA	Xilinx Alveo U280 [91]
DSP blocks	8490
BRAM/URAM	6.6MB (1487 blocks)/34.5MB (960 blocks)
HBM	8GB, 460GB/s, 32 channels
DDR4	32GB, 38.4GB/s, in 2 banks
Host	AMD Ryzen Threadripper PRO 3975WX (32 cores)
	512GB RAM, Ubuntu 18.04.6 LTS
Design SW	Xilinx Vivado HLS, Vitis 2019.2
Run-Time	Xilinx XRT 202020.2.9.317
GPU	Nvidia Tesla V100 PCIe [53]
Global Mem.	16GB HBM2, 900GB/s
Host	Intel Xeon Gold 6252 @2.10GHz (48 cores)
	256GB RAM, Ubuntu 18.04.3 LTS
Compilers, OS	nvcc CUDA 10.0.130, Debian 9.11

ADI 2D: FP32 (Figure 4.5, Runtimes in Seconds)

Mesh	32^2	48^2	64^2	80^2	96^2	112^2	128^2
GPU-1500B	0.051926	0.100185	0.164678	0.252945	0.352306	0.478392	0.608963
GPU-3000B	0.08929	0.185628	0.311998	0.490501	0.694886	0.946668	1.21891
FPGA-1500B	0.011768	0.024065	0.045064	0.06177	0.087886	0.119968	0.15678
FPGA-3000B	0.020959	0.044525	0.084882	0.118695	0.169448	0.23099	0.304469
Pred-1500B	0.011426	0.023794	0.040598	0.061838	0.087514	0.117626	0.152175
Pred-3000B	0.020193	0.043520	0.075666	0.116632	0.16641	0.22502	0.292449

ADI 2D: FP64 (Figure 4.5, Runtimes in Seconds)

Mesh	32^2	48^2	64^2	80^2	96^2	112^2	128^2
GPU-1500B	0.083552	0.180618	0.309131	0.486596	0.688309	0.959437	1.262698
GPU-3000B	0.158329	0.35355	0.627219	0.975146	1.3695	1.908415	2.448809
FPGA-1500B	0.032777	0.070323	0.124069	0.183378	0.260491	0.351374	0.459245
FPGA-3000B	0.0598	0.13201	0.236424	0.352789	0.503768	0.682056	0.894456
Pred-1500B	0.032033	0.06821	0.117826	0.180883	0.25738	0.347316	0.450693
Pred-3000B	0.0587	0.12821	0.224493	0.34755	0.49738	0.673983	0.87736

ADI 3D: FP32 (Figure 4.6, Runtimes in Seconds)

Mesh	32^3	80×32^2	48^3	80×64^2	80^3	96^3
GPU-24B	0.033485	0.064337	0.078677	0.207939	0.318074	0.534816
GPU-72B	0.068051	0.159311	0.209396	0.605547	0.966475	1.605493
FPGA-24B	0.014435	0.031197	0.036908	0.09639	0.145938	0.245777
FPGA-72B	0.035474	0.073093	0.0942	0.268703	0.418207	0.712403
Pred-24B	0.013710	0.030467	0.036084	0.094831	0.142153	0.237960
Pred-72B	0.029860	0.070842	0.090591	0.256332	0.394499	0.674013

ADI 3D: FP64 (Figure 4.6, Runtimes in Seconds)

Mesh	32^3	80×32^2	48^3	80×64^2	80^3	96^3
GPU-24B	0.047983	0.109056	0.143062	0.406282	0.630805	1.090251
GPU-72B	0.125366	0.307859	0.410969	1.210522	1.894097	3.200741
FPGA-24B	0.031326	0.07085	0.087803	0.240179	0.371648	0.626418
FPGA-72B	0.078996	0.180131	0.238624	0.689622	1.089421	1.847857
Pred-24B	0.030766	0.070762	0.087616	0.240405	0.366186	0.620672
Pred-72B	0.074453	0.179989	0.235072	0.677312	1.048853	1.80032

ADI 2D: Thomas-PCR, FP32 (Figure 4.7, Runtimes in Seconds)

Mesh	256^2	384^2	512^2	640^2	768^2	896^2
GPU-12B	0.089786	0.142	0.19506	0.2532	0.321498	0.387086
GPU-60B	0.137726	0.250878	0.383688	0.568991	0.795465	1.049155
GPU-180B	0.270218	0.567759	0.982022	1.548083	2.197039	2.986251
FPGA-12B	0.015729	0.03157	0.051094	0.077999	0.109692	0.148295
FPGA-60B	0.060961	0.13298	0.231108	0.360772	0.514358	0.701884
FPGA-180B	0.174262	0.386244	0.681082	1.067558	1.525821	2.085651
Pred-12B	0.015261	0.030717	0.049586	0.075453	0.106781	0.143570
Pred-60B	0.058952	0.129021	0.224349	0.34852	0.499997	0.678781
Pred-180B	0.168178	0.374781	0.661256	1.031186	1.483037	2.016808

ADI 2D: Thomas-Thomas, (Figure 4.7, Runtimes in Seconds)

Mesh	256^2	384^2	512^2	640^2	768^2	896^2
GPU-12B	0.089786	0.142	0.19506	0.2532	0.321498	0.387086
GPU-60B	0.137726	0.250878	0.383688	0.568991	0.795465	1.049155
GPU-180B	0.270218	0.567759	0.982022	1.548083	2.197039	2.986251
FPGA-12B	0.016794	0.033331	0.051978	0.079304	0.111366	0.150252
FPGA-60B	0.062061	0.13553	0.232126	0.36213	0.516121	0.703584
FPGA-180B	0.175236	0.389155	0.68165	1.069117	1.527731	2.087249
Pred-12B	0.015914	0.032064	0.053674	0.080746	0.11328	0.151274
Pred-60B	0.059605	0.130368	0.228437	0.353813	0.506496	0.686485
Pred-180B	0.168832	0.376128	0.665344	1.03648	1.489536	2.024512

SLV: Batched 40×20 (Figure 4.8, Runtimes in Seconds)

Batch	30	300	3000
GPU	0.008342	0.014189	0.031037
FPGA	0.001705	0.004655	0.033508
FPGA_Pred	0.001357	0.004187	0.032477

SLV: Batched 100×50 (Figure 4.8, Runtimes in Seconds)

Batch	30	300	3000
GPU	0.060663	0.202378	1.20224
FPGA	0.053073	0.23719	2.077623
FPGA_Pred	0.052469	0.236359	2.075259

C.3 Chapter 5 Runtimes

Experimental systems specifications.

FPGA	Intel PAC D5005 [29]
DSP blocks	5760
MLABs / M20K	7.6MB / 29.3 MB
DDR4	64GB, 76.8GB/s, in 4 banks (1 channel/bank)
Host	Intel Xeon Platinum 8256 @3.8GHz
	(16 CPUs, 4 cores each)
	1559 GB RAM, Ubuntu 18.04.6 LTS
Design SW	Intel oneAPI 2021.4.0, Intel Quartus software 19.2
board_variant	pac_s10
GPU	Nvidia Tesla V100 PCIe [53]
Global Mem.	16GB HBM2, 900GB/s
Host	Intel Xeon Gold 6252 @2.10GHz (48 cores)
	256GB RAM, Ubuntu 18.04.3 LTS
Compilers, OS	nvcc CUDA 10.0.130, Debian 9.11

RTM forward-pass: Batched (Figure 5.1, Runtimes in Seconds)

Mesh	10^3	16^3	22^3	28^3	34^3	40^3
GPU-10B	0.011478	0.014003	0.033032	0.058117	0.098145	0.189857
GPU-100B	0.035713	0.080376	0.209297	0.408905	0.848501	1.723459
FPGA-10B	0.011721	0.023837	0.043807	0.072718	0.112303	0.164298
FPGA-100B	0.094341	0.211727	0.400258	0.703664	1.057610	1.559980
Pred-10B	0.011322	0.023992	0.044029	0.073109	0.112904	0.165090
Pred-100B	0.094206	0.211527	0.399843	0.675899	1.056439	1.558206

ADI 2D on Intel FPGAs: FP32(Figure 5.1, Runtimes in Seconds)

Mesh	32^2	40^2	48^2	56^2	64^2
GPU-800B	3.938412	5.624536	7.167415	9.374078	11.206901
GPU-4000B	13.515752	21.061396	29.08813	40.131187	50.196379
GPU_opt_est-800B	1.68678	2.383387	3.011662	3.926800	4.647422
GPU_opt_est-4000B	5.560471	8.74728	12.02255	16.62653	20.63857
FPGA-800B	1.08793	1.62754	2.28043	3.04603	3.924
FPGA-4000B	4.63369	7.16889	10.2588	13.9052	18.1093
Pred-800B	1.066900	1.609792	2.265722	3.034692	3.916701
Pred-4000B	4.613220	7.150917	10.24494	13.8952	18.10198