

Design Space Exploration of Concurrency Mapping to FPGAs in Weather and Climate Applications with Xilinx SDSoC OpenCL, SDSoC C++ and Vivado

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Moteb Salem Alghamdi

Department of Computer Science
University of Manchester

Contents

Abstract	14
Declaration	16
Copyright	17
Acknowledgements	18
1 Introduction	20
1.1 Motivation	21
1.2 Research Questions	24
1.3 Thesis Contributions	25
1.4 Thesis structure	26
1.5 Publications	27
2 Background and Related Work	28
2.1 Field Programmable Gate Arrays (FPGA)	28
2.2 High Level Synthesis Tools	29
2.3 SDSoC HLS	30
2.3.1 SDSoC OpenCL	32
2.3.2 SDSoC C++	34
2.4 Xilinx Vivado	37
2.5 Vivado and SDSoC OpenCL/C++ Optimisation Strategies	40
2.5.1 Kernel Computation and Data-movement Optimisation Attributes and Pragmas	42
2.6 HPC-based Benchmark Applications	45
2.6.1 Shallow Water Dynamics Model	45
2.6.2 LFRic Weather and Climate Model mini-app	53

2.6.3	Summary	55
2.7	Related Work	55
2.7.1	Single HPC-Kernel accelerators	56
2.7.2	Exploratory studies	58
2.7.3	Comparison and Survey studies	61
2.7.4	Related Work Summary	63
3	Research Methodology and Study Experiments Setup	64
3.1	Target FPGA Hardware	64
3.2	System and Experimental Setup	67
3.3	Exploratory Study	67
3.3.1	Exploratory Study (1): SWM Concurrency Mapping Exploration	67
3.3.2	Exploration Study (2): <i>MatVec</i> Kernel with SDSoC OpenCL/C++ and Vivado HLS	68
3.4	Comparison Study	69
3.4.1	Comparison Study (1): SWM Implementations In SDSoC OpenCL versus Vivado	69
3.4.2	Comparison Study (2): <i>MatVec</i> Kernel implementations in SDSoC OpenCL and C++ Versus Vivado HLS	69
3.4.3	Comparison Study Metrics	70
3.5	Summary	71
4	Exploratory Study (1) Part One: L100 kernel Concurrency Mapping	72
4.1	L100 Concurrency and Coding Options	72
4.2	Study Setup	74
4.3	L100 kernel mapping using SDSoC OpenCL	75
4.3.1	SDSoC OpenCL L100 Initial Implementation	76
4.3.2	Mapping Mechanism Experiments	80
4.3.3	Instruction-level parallelism	81
4.3.4	Data parallelism	82
4.3.5	Functional parallelism	85
4.3.6	Discussion	87
4.4	L100 kernel mapping using Vivado HLS	89
4.4.1	Vivado Initial Implementation	90
4.4.2	Mapping Mechanism Experiments	96
4.4.3	Instruction-level parallelism	97

4.4.4	Functional Parallelism	97
4.4.5	Discussion	102
4.5	Summary	108
5	Exploratory Study (1) Part Two: SWM Multiple-Kernels Mapping	109
5.1	SWM Kernels Mapping Using SDSoC OpenCL	110
5.1.1	Optimise the SWM SDSoC OpenCL kernels	110
5.1.2	Exploring the problem size	113
5.1.3	Kernel-to-kernel communication exploration	113
5.2	SWM Application Mapping Using Vivado	122
5.2.1	Optimise the SWM Vivado kernels	122
5.2.2	Finding the problem size	123
5.2.3	Kernel-to-Kernel Communication Exploration	123
5.3	Summary	125
6	Comparison Study (1): SWM Implementations In SDSoC OpenCL Versus Vivado	127
6.1	L100 concurrency mapping comparison	127
6.1.1	Performance Analysis	128
6.1.2	Resource Usage Analysis	134
6.2	Multiple kernel mapping comparison	134
6.2.1	Performance Analysis	135
6.2.2	Resource Usage Analysis	137
6.3	Development Effort and Hardware Level of Expertise	137
6.4	Summary	138
7	Exploration Study (2): MatVec Kernel with SDSoC OpenCL, SDSoC C++ and Vivado	140
7.1	MatVec Reference Implementation	141
7.2	MatVec Xilinx Vivado Design	141
7.2.1	MatVec Xilinx Vivado Design Overview	142
7.2.2	MatVec Xilinx Vivado Kernel Code	142
7.2.3	MatVec Vivado Hardware Design	147
7.2.4	MatVec Vivado CPU code	151
7.3	MatVec OpenCL design	156
7.3.1	MatVec SDSoC OpenCL Kernel code	157

7.3.2	<i>MatVec</i> SDSoC OpenCL, Hardware Design	162
7.3.3	<i>MatVec</i> SDSoC OpneCL, Host Code	165
7.4	<i>MatVec</i> SDSoC C++, design	170
7.4.1	<i>MatVec</i> SDSoC C++, Kernel code	171
7.4.2	<i>MatVec</i> SDSoC C++, Hardware Design	176
7.4.3	<i>MatVec</i> C++ Host Code	180
7.4.4	Other SDSoC OpenCL and C++ <i>MatVec</i> Design Alternatives	182
7.5	Summary of <i>MatVec</i> Exploration Study	187
7.6	Summary of the Two Exploratory Studies	189
7.6.1	Single kernel	189
7.6.2	Multiple kernels	189
8	Comparison Study (2): <i>MatVec</i> Kernel implementations in SDSoC OpenCL and SDSoC C++ Versus Vivado HLS	192
8.1	Performance Analysis	193
8.1.1	Computation Flop Rate (Gflop/s)	193
8.1.2	Application Runtime Considerations	198
8.2	Resource Usage Analysis	201
8.3	Data Movement Analysis	202
8.3.1	Matvec CPU Implementation Comparison	203
8.4	Summary of the <i>MatVec</i> Comparison Study	204
8.5	Summary of the Two Comparison Studies	205
8.5.1	Single kernel	205
8.5.2	Multiple kernels	205
9	Conclusions and Future Work	207
9.1	Review of Thesis Research Questions	207
9.2	Summary of Contributions	215
9.3	General Recommendations	216
9.4	Limitations and Future Work	217
	Bibliography	219
A	The Shallow Water Model Source Code	233
A.1	SWM Nine Kernels Host Source Code	233
A.2	SWM Nine Kernels Source Code	240

B Block designs

249

Word Count: 46498

List of Tables

2.1	Summary of the current available HLS tools and their properties: Availability Some of the tools are made freely accessible by the developers, while others require a licence. Target architectures There are two main FPGA board producers, Xilinx [Wind] and Altera [Wina]; Some target a specific FPGA board; while others target both FPGA platforms. Computation type demonstrates whether a specific tool supports the dataFlow or controlFlow paradigm [SMB ⁺]. Input Language describes the design entry language of the tool. [KHZ16] [NSP ⁺ 16]. Information in the empty cells is not available.	31
2.2	The properties details of the available Data movers engines in the SD-SoC C/C++ approach.	36
2.3	SDSoC OpenCL/C++ and Vivado HLS Kernel Computation and data-movement Optimisations Attribute and Pragmas.	42
3.1	Xilinx Zynq UltraScale+ MPSoC ZCU102 board Available Resources Count.	66
4.1	The performance effects of each optimisation on the L100 kernel in SDSoC OpenCL. (Seq) no optimisations; (Max) max DDR memory ports; (BRAMs) use of BRAM memories; (B) use burst mode; (AP) use array partitioning	76
4.2	ILP mapping performance of the L100 kernel in SDSoC OpenCL. (P) pipelining, (a) or (b) code options (a) or (b) in Figure 4.1; (U) un-rolling.	81
4.3	Data parallelism performance of the L100 kernel in SDSoC OpenCL: NDRange. (N) NDRange kernel; (CU) No. of compute units; (WG) No. of work-groups.	84

4.4	Functional parallelism performance of the L100 kernel in SDSoC OpenCL: Dataflow. (DF) dataflow ; (F) apply DF to function code style; (L) apply DF over loops; (M) A kernel per operation; (P) pipeline; (U) unrolling; N NDRange kernel.	85
4.5	Comparison of the number of compute units (CUs), the number of command queues (CQs), and loops, and the coding difficulty (score from 1 to 5) and resource usage of different mechanisms implemented on the L100 kernel in SDSoC OpenCL.	88
4.6	The performance figures of each optimisation on the L100 kernel in Vivado HLS. (Seq) no optimisations; (BRAMs) use of BRAM memories; (B) use burst mode; (AP) use array partitioning	94
4.7	The resource usage figures of each optimisation on the L100 kernel in Vivado HLS. (Seq) no optimisations; (BRAMs) use of BRAM memories; (B) use burst mode; (AP) use array partitioning.	95
4.8	The performance figures of the ILP and Functional parallelism mapping of the L100 kernel in the Vivado approach. (P) pipelining; (a) code option (a) in Figure 4.1; (DF) dataflow, (F) apply DF to function code style; (M) A kernel per operation. The blank entries in this table mean that the information is not available to be reported. The speed up is relative to the Vivado L100-seq initial implementation.	98
4.9	The resource usage figures of the ILP and Functional parallelism mapping of the L100 kernel in the Vivado approach. (P) pipelining; (a) code option (a) in Figure 4.1; (DF) dataflow, (F) apply DF to function code style; (M) A kernel per operation.	99
4.10	Comparison of the programming difficulty levels (score from 1 to 5) between the different mechanisms implemented on the L100 kernel in the Vivado HLS approach.	107

5.1	The compute time detail for the three SDSoC OpenCL SWM 5 Kernels optimized versions implementations Versus the SDSoC OpenCL SWM 5 Kernels un-optimized version. All versions implemented on ZCU102 FPGA board with a clock frequency of 200 MHz (The maximum possible frequency). In the "DDR" implementation the host will explicitly start kernels when previous kernels have completed, so there is no kernel to kernel marshalling here. Un-optimised SWM kernels implementations are similar to the L100_seq implementation in Chapter4.	117
5.2	Resource Usage Figures of the SDSoC OpenCL SWM Five Kernels implementations.	122
5.3	Performance Figures of the Vivado SWM 5 kernels implementation. .	125
5.4	Resource Usage Figures of the Vivado SWM Five Kernels implementation.	125
6.1	The application runtime details of the best L100 SDSoC OpenCL and Vivado implementations in Chapter 4.	128
6.2	Total design resource usage figures for the best L100 SDSoC OpenCL and Vivado implementations in Chapter 4.	135
6.3	The application runtime details for the SWM five Kernels implementations in SDSoC OpenCL and Vivado approaches in Chapter 5. . . .	135
6.4	Total design resource usage figures for the best SWM five Kernels implementations in SDSoC OpenCL and Vivado approaches in Chapter 5.	137
7.1	Latency figures comparison between the SDSoC OpenCL <i>MatVec</i> Dataflow implementation Versus the SDSoC OpenCL <i>MatVec</i> design with no Dataflow in terms of Latency (clock cycles) figures. The test design is One <i>MatVec</i> block with 26 cells.	184
7.2	Calc function latency breakdown of the SDSoC OpenCL <i>MatVec</i> Dataflow implementation Versus the SDSoC OpenCL <i>MatVec</i> design with no Dataflow.	185
8.1	Performance details for the best Matvec implementations from Vivado, SDSoC OpenCL and SDSoC C++. (*: performance results for a smaller, 1 block/26 cell versions)	199
8.2	The best implementations resource usage figures for the <i>MatVec</i> IP block and the total system design.	201

8.3	Rates of data movement for the best implementations	202
8.4	Comparison of ZU9 FPGA double-precision Vivado, OpenCL and C++ matrix-vector performance implementations with Intel multicore CPU performance	203

List of Figures

2.1	SDSoC Design Flow [SG20].	32
2.2	OpenCL memory model in Xilinx Zynq UltraScale+ MPSoCZCU102.	33
2.3	“SDSoC data motion network components. PS means the processing system CPU part. Acc means the accelerator par which is the FPGA (A) is the system port type. (B) are the data mover engines. (C) is the accelerator interface port type” [Xil21h].	35
2.4	An example of IP Repository in the Vivado IP Catalog.	38
2.5	A full Vivado system design example with an IP kernel integrated to the design.	39
2.6	An example of the Vivado address editor.	39
2.7	An Arakawa-C grid mapping for the shallow water variables [Sad75, Pap12].	47
2.8	SWM nine OpenCL kernels representation.	50
2.9	Halo regions representation in the SWM arrays elements.	51
2.10	A highlight of the data flow of the five main kernels in the SWM coding algorithm.	52
2.11	“cubed-sphere mesh as used in GungHo with 12×12 subdivisions per face, referred to as a C12 mesh. This gives 864 columns of cells” [AFH ⁺ 19].	53
3.1	Zynq UltraScale+ MPSoC Top-Level Block Diagram [Xil20].	65
4.1	Pseudocode for the <i>L100</i> kernel coding options. A- Wrap the kernel operations with one <i>for</i> loop. B- Wrap each operation in a loop. C- Wrap each operation in a <i>function</i>	73
4.2	Effects of optimisation on resource utilisation.	78
4.3	Bar graph showing the resource utilisation of the optimised concurrency mapping implementations listed in Section 4.3.1.	82

4.4	Schedule view of the 11 created CUs for L100-DF-F implementation from the SDSoC OpenCL build reports.	86
4.5	Bar graph showing the resource utilisation of the optimised concurrency mapping implementations listed in Section 4.3.2.	87
4.6	Vivado L100 kernel IP block.	92
4.7	Vivado L100 Initial implementation system design	93
4.8	Vivado L100-DF IP Block.	101
4.9	Vivado L100-DF-F-1-BRAM-Block implementation system design. .	103
4.10	Vivado L100-DF-F-7-BRAM-Blocks implementation system design .	104
4.11	Vivado L100-M-P-4-BRAM-Blocks implementation system design. .	105
5.1	The implementation system design for SDSoC OpenCL Five kernels with DDR mechanism for kernel-to-kernel communication implementation system design	114
5.2	OpenCL Pipe IP Block example.	115
5.3	The implementation system design for the SDSoC OpenCL Five kernels with Pipes (Version 1) mechanism for kernel-to-kernel communication implementation system design	119
5.4	The implementation system design for the SDSoC OpenCL Five kernels with Pipes (Version 2) mechanism for kernel-to-kernel communication implementation system design	121
5.5	The implementation system design for the Vivado Five kernels with one external BRAM method for kernel-to-kernel communication . . .	124
7.1	Overview of the <i>MatVec</i> Vivado HLS design.	142
7.2	The generated <i>MatVec</i> Vivado IP block.	147
7.3	Overview a Vivado two <i>MatVec</i> IP blocks design.	148
7.4	Address map example for two Vivado <i>MatVec</i> design.	151
7.5	Example of the register map of the Vivado <i>MatVec</i> IP block.	155
7.6	Overview of the SDSoC OpenCL <i>MatVec</i> design.	158
7.7	Overview of the SDSoC OpenCL one <i>MatVec</i> IP block system design	163
7.8	OpenCL <i>MatVec</i> IP block.	164
7.9	Overview of the SDSoC OpenCL four <i>MatVec</i> IP blocks system design	166
7.10	Overview of the SDSoC C++ <i>MatVec</i> design.	171
7.11	Overview of the SDSoC C++ one <i>MatVec</i> IP block system design . . .	177
7.12	The SDSoC C++ <i>MatVec</i> generated IP block.	178

7.13	The flops timeline of the Calc function in the SDSoC OpenCL <i>MatVec</i> no-Dataflow implementation.	185
7.14	The flops timeline of the Calc function in the SDSoC OpenCL <i>MatVec</i> Dataflow implementation.	186
8.1	Performance of the Vivado Matrix-vector kernel designs at 310Mhz, as the number of blocks and cells-per-block varies. Performance figures here are the kernel "compute time only"	194
8.2	Performance of the OpenCL Matrix-vector kernel designs at 200Mhz as the number of blocks and cells-per-block varies. Performance figures here are the kernel "compute time only"	196
8.3	Performance of the C++ Matrix-vector kernel designs at 150Mhz, as the number of blocks and cells-per-block varies. Performance figures here are the kernel "compute time only"	197
B.1	Overview of the SDSoC C++ six <i>MatVec</i> IP blocks system design . .	250

Abstract

DESIGN SPACE EXPLORATION OF CONCURRENCY MAPPING TO FPGAS IN WEATHER AND CLIMATE APPLICATIONS WITH XILINX SDSoC OPENCL, SDSoC C++ AND VIVADO

Moteb Salem Alghamdi

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2022

Recent years have seen increased interest from the HPC community in Field Programmable Gate Arrays (FPGAs) as an alternative/additional accelerator. This has been largely due to the slowdown in the transistor scaling and the difficulty of gaining performance improvement and energy efficiency from the current processing solutions. General (scientific) software programmers have shied away from the FPGA technology because of their perceived lack of programmability. However, various academic and commercial vendors have now developed High-Level Synthesis (HLS) tools, such as Xilinx SDSoC OpenCL, SDSoC C++, Vivado HLS, Intel Altera SDK and solutions from Maxeler, which enable the generation of FPGA hardware configurations from higher-level descriptions. Even though HLS tools aim to minimize the hardware knowledge gap between the software programmers and FPGAs, HLS tool programming methodologies are still challenging for the software programmers aiming to achieve high performance. These HLS tools impose many choices for mapping the concurrency in HPC applications to FPGAs. The choices are complex as they include different options available at the programming language level and the HLS tool level for designing the host code, the kernel code, and the FPGA hardware itself. Furthermore, a wide choice of optimization methods controls the final *design* of the FPGA hardware. The many options and different parameter settings available can severely affect a design's performance and also the programmer's productivity and lead to a large *design space exploration* problem. The HPC software programmer has to spend much time finding the appropriate options and parameter settings that provide the best

possible design. Furthermore, choosing a suitable HLS tool from those available is another complexity in utilizing HLS tools. This thesis explores and compares the options and techniques for mapping the concurrency levels of two weather and climate applications using two high-level HLS tools, Xilinx SDSoC OpenCL and SDSoC C++, and a low-level HLS tool, Xilinx Vivado HLS, to a single Xilinx Ultrascale+ FPGA board. Two exploratory and two comparison studies have been conducted, involving many experiments, to provide insight into the best mapping techniques for performance, resources usage, and programmability in single and multiple kernel solutions. In the exploratory studies of the design space, data was collected from various implementations and configurations of the two weather and climate applications. This data is then utilized and analyzed, using multiple metrics, in the two comparison studies, providing insights for traditional HPC programmers considering using FPGAs in their applications and also contributing evidence to support the possible future development of an efficient methodology for their use.

Declaration

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

Copyright

- i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.
- ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.
- iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.
- iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see <http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487>), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see <http://www.manchester.ac.uk/library/aboutus/regulations>) and in The University’s policy on presentation of Theses

Acknowledgements

بسم الله الرحمن الرحيم

الحمد لله الذي بنعمته تتم الصالحات الحمد لله كما ينبغي لجلال وجهه وعظيم سلطانه على توفيقه وتيسيره لانهاء هذه الرسالة.

Words cannot describe my gratitude to my supervisor Graham Riley for his dedicated support, guidance, and help. Graham continuously provided encouragement, and he was always there to assist in any way he could, throughout the ups and downs of my PhD journey and before in my master's degree. Thanks to him for enlightening my knowledge with the fruitful and engaging discussions that shaped this thesis.

I want to thank my friends at the Advanced Processor Technologies group: Abdullah Khalufa, Ahmed Alghamdi, Guillermo, Konstantinos Iordanou, and Swapnil, for their kind help, support, and companionship that have made my study and life in the UK a wonderful time. A special thanks go to Mike Ashworth and Andrew Attwood for the help and technical support.

I am indebted to my parents: my dad (Salem), who supported me, stood by me, and sadly died before he could see me finishing my PhD. I devoted this work to him and my mom (Ajab), who gave me her love, help, and prayers and made numerous sacrifices to get me to this point in my life. I am also grateful to my wife (Abrar) for her patience, love, and support through all these years. She devoted her life to the success of me and of the kids, Turki and Reema. Words cannot describe all the sacrifices those three made to help me get through this journey.

My gratitude extends to my brothers: Abdullah, Hamdan, Ahmed, Talal, Ali, and sisters: Hanan, Dalal, Norah. Without their tremendous understanding and encouragement in the past few years, it would be impossible for me to complete my study.

I am thankful to my closest friends (Hamza, Khaled, Anwar, Fahad), who always provided support, advice, and help. I also extend my thanks to all Saudi students fellows who I had the pleasure to know during my time in the UK.

I would also like to express my sincere gratitude to my country Saudi Arabia and Taibah University, for the funding opportunity to undertake my studies and their support throughout this journey.

Chapter 1

Introduction

In recent years, industry and academia have been exploring an alternative computational form for their application needs [HP19] due to the slowdown in the transistor scaling and the difficulty of gaining performance improvement from the current processing solutions. The continued transistor scaling has not been delivering better energy scaling; therefore, accelerators (heterogeneous computing) became the new norm for increasing performance. In addition, energy efficiency is now a priority concerning the high performance community [SP11]. In current dominant HPC architectures, CPU modules are utilized, coupled with bus-attached GPU accelerators to boost the performance of the HPC workloads. Undeniably, GPUs provide unbeatable performance for the HPC workloads due to their high memory bandwidth, however, they are power-hungry, and their performance is limited to specific domains [MV14].

An alternative accelerator that the HPC community has an interest in is Field Programmable Gate Arrays (FPGAs) due to their better performance-per-watt over other technologies such as GPUs [MV14]. Recently, FPGAs have been explored for their potential to accelerate traditional HPC applications [DRP11, HM17, Bro19, BD19, Bro21, BKB21, MPK21, KPJ⁺21, KSFNA21, KPK⁺22]. They have been deployed in a range of HPC applications, for example, in large-scale scientific application domains [DRP11, HM17], including weather and climate applications [GFY⁺14, ARAM19, VN14, SKN⁺16], fluid computations [SHY14, GFY⁺14, KSFNA21], SLAM algorithms [AEB⁺16] and linear algebra, e.g. Matrix Multiplication algorithms [DVKG05, OPAM21]. FPGAs are also emerging in large-scale computers on the path to Exascale computing [HM17], such as in the EUROEXA project [Pro20].

Despite the performance-per-watt that FPGAs offer and the growing interest in their use in HPC applications, they have been, until recently, restricted in their use to a

narrow group of hardware programmers. This was because the low-level nature of FPGAs meant that significant programmer effort was needed to achieve performance levels comparable with traditional architectures. Traditionally, implementing applications on FPGAs has been delivered through the use of low-level hardware design languages such as VHDL [Ash10] and Verilog [TM08]. These conventional methods require a high level of hardware design expertise [BRS13]. A hardware design would involve building using low-level blocks such as gates, registers and multiplexers, in addition to hand crafting the application’s datapath, the computational pipeline of hardware modules required, memory management, communication interfaces, and the specification of the behaviour required of the datapath in every register and model in the design. These programming languages require extensive low-level hardware knowledge and significant development time and effort [BRS13].

General (scientific) software programmers have shied away from the FPGA technology because of the FPGAs’ lack of programmability. However, various academic and commercial vendors have now developed High-Level Synthesis (HLS) tools [CLN⁺11] [KHZ16] which enable the generation of FPGA hardware configurations from higher-level descriptions, based on such languages as C/C++, OpenCL and Java. HLS tools convert an application’s algorithm into a low-level register-transfer level (RTL) description. Several compiler frameworks and languages have been proposed for easy-to-use FPGA programming methods. All aim for raising the level of abstraction at which the software programmer can implement an application to be compiled down to the equivalent of VHDL or Verilog specifications (and then to RTL). Examples of HLS tools are Xilinx Vivado HLS [VH20], Intel Altera SDK [Int20], Xilinx SDSoC [Xil19] and Maxeler [Max20].

1.1 Motivation

HLS tool vendors have adopted high-level languages to simplify the FPGA’s programmability, specifically targeting the HPC software programmers who come new to FPGAs. Even though HLS tools aim to minimize the hardware knowledge gap between the software programmers and FPGAs, HLS tool programming methodologies are still challenging for the software programmers aiming to achieve high performance. Recent studies have explored these challenges and have been evaluating the HLS tools’ techniques and methods [NSP⁺16, CFH⁺18, PKB⁺16], and developing improved optimisation strategies and programming methodologies [SVK, BRS13, GLN⁺14, CBM⁺18]

to enhance the programmability experience of users of the HLS tools aiming to achieve high performance. Moreover, researchers from different HPC areas have contributed to the advancement of the HLS tools programmability by, for example, creating (specific) fine-tuned accelerator designs [GBLS16, SKN⁺16, ZPM18], evaluating memory design solutions [GCDJ19, Sve16, LWY⁺17] and developing optimisation methods [CFH⁺18, ARAM19, KPZ⁺16].

However, other issues remain outstanding in the effort to improve FPGA programmability in HPC applications, including in weather and climate HPC applications. Projects such as EUROEXA [Pro20] have been investigating the use of HLS tools to improve the accessibility of FPGAs, in the context of exascale computing, to the weather and climate application programmers as well as other application domains.

HPC software programmers already face a complex design problem when choosing how best to map the concurrency in their applications to the heterogeneous hardware resources in traditional HPC systems. Concurrent designs are typically mapped into high-level language extensions such as MPI to exploit parallelism over the shared memory nodes typical of current HPC architectures, OpenMP to exploit shared memory parallelism within a node and OpenACC, OpenCL or CUDA to exploit GPU accelerators. In each of these cases, the target hardware is essentially fixed. With the introduction of the FPGAs and HLS tools, HPC programmers can control the design of (some of) the actual hardware itself. This makes the concurrency mapping problem much more complex since the choices now include language options which control the *design* of the FPGA hardware.

For example, OpenCL is portable and easy-to-use [MGMG11] and is being explored by the HPC community for the acceleration of applications on FPGAs [SVK18, Zoh18, WHU18]. However, although OpenCL compilers for FPGAs generate functionally correct hardware designs, achieving high performance remains a challenge. The many options and different parameter settings available can affect a design's performance and also the programmer's productivity. The different options that the OpenCL HLS tool provides for use in writing the kernel code and the different optimisation implementation options lead to a **design space exploration problem**. The HPC software programmer would spend much time finding the options and parameter settings that provide the best possible design. In addition, current HPC weather and climate applications consist of multiple kernels which lead to multiple levels of concurrency in the algorithms; therefore, porting such algorithms effectively to FPGAs requires another

phase of exploration for the software programmer. Several aspects have to be considered in consolidating the final FPGA design, such as the concurrency mapping choices, the number of the kernels to map to the FPGAs in the computer system, the size of the problem which can execute on the FPGAs, the data movement design across the computer system, and the memory hierarchy optimisation. The software programmer has to have a rationale for choosing among all these options.

HPC software programmers also face another challenge related to choosing a suitable HLS tool from those available. A few efforts have been made to address this challenge, such as the study in [NSP⁺15]. Choosing the right HLS tool is another complexity that limits the accessibility of the HLS tools. HLS tools vary in terms of the required low-level hardware knowledge, level of hardware control, the design flow, the programming language support they provide, and the learning time they require. Those differences contribute to the productivity when using the chosen tool, and the possible performance that can be achieved. For example, the use of the Xilinx High-Level Productivity Design Methodology approach Xilinx Vivado tool [VH20] requires the acquisition of deep hardware design skills to benefit from its capabilities fully, and the programmer needs to perform manual optimisations, that involve several low-level design stages, manual creation of the hardware solution components while taking care of the interconnection between the components. In other, relatively higher-level, tools such as Maxeler, Xilinx SDSoC and SDAccel, programmers are required to directly control fewer hardware aspects, such as the number of computing units to be used, the number of memory ports to exploit and the use of some low-level optimisations through high-level language specifications (e.g. through the use of directives).

Therefore, choosing and using an appropriate HLS methodology for the hardware design is a research problem, and there is a need for an exploration study from the high-level scientific programmer's perspective across the different tools and a comparison between the HLS tools when developing solutions for the same algorithms to examine their differences in terms of programmability and performance.

This thesis presents two exploratory studies that collect data from different weather and climate applications implemented on different HLS approaches independently and two comparison studies that compare, based on that data gathered, between and across the HPC applications and HLS approaches. These studies are intended to contribute to the analysis and evaluation of the challenges regarding the HLS tools programmability and performance. Given the large scope of the problem, we focus on example

applications from the weather and climate domain targeting a specific Xilinx system-on-chip FPGA. The two applications from the weather and climate domain are a well studied, traditional finite difference Shallow Water Dynamics Model (SWM) [[Sad75](#)] and a modern finite element example of the LFRic atmospheric weather forecasting model being developed at the UK Met Office [[Off21](#)]. For the HLS tools, we focus on the Xilinx High-Level Productivity Design Methodology approach Xilinx Vivado tool and the relatively higher level Xilinx SDSoC tool supporting both OpenCL and C++.

In the two exploratory studies, a large number of experiments have been conducted. The first one is conducted, in two parts, to explore the concurrency mapping problem of single kernels in the SWM application (in part one) and across multiple kernels in the application (in part two) utilising the (high-level) SDSoC OpenCL and the (lower-level) Xilinx High-Level Productivity Design Methodology Vivado approach. In the second exploratory study, several experiments have been conducted to explore trying to replicate an existing FPGA design for the LFRic application created using the (lower-level) HLS tool Xilinx High-Level Productivity Design Methodology Vivado approach in the (higher-level) HLS tools, SDSoC OpenCL and SDSoC C++). The exploratory studies collect performance and resource usage data for the different implementation options (quantitative data). In addition, in the studies we also investigate the different options available in the HLS tools and in the programming languages used, and comment on the trade-offs involved in their use, including the impact on programmability (qualitative data).

In the two comparison studies, two evaluations, based on a systematic comparison between the Vivado HLS, SDSoC OpenCL and SDSoC C++ tools, are conducted using multiple metrics based on the quantitative and qualitative data collected from the exploratory studies. One comparison study compares the data of the SWM implementations (single and multiple kernel examples) collected using the Vivado HLS and the SDSoC OpenCL tools. The other compares the results of the implementations from the LFRic exploration studies using the Vivado HLS, SDSoC OpenCL and SDSoC C++ tools. Finally, some conclusions are drawn from the outcomes *across* both studies.

1.2 Research Questions

As described above, this thesis's main objective is to contribute to the accessibility of the FPGA to (traditional high-level) software programmers for accelerating weather

and climate application. The research challenges mentioned above concern the lack of exploratory and comparison studies for mapping concurrency levels in scientific applications, specifically weather and climate HPC applications, to FPGAs.

The main research questions (RQs) that this thesis aims to answer are:

- What are the technology options from the FPGA level and the HLS tool (SDSoC OpenCL and Vivado HLS) level for mapping the concurrency within a single HPC kernel and in the case of multiple HPC kernels?
- Can the use of high-level optimization techniques in SDSoC OpenCL and SDSoC C++ match the design choices and performance achievable from the use of the (lower-level) manual optimizations in Xilinx High-Level Productivity Design Methodology Vivado approach ?
- What can be said about the best mapping technology options suitable for HPC application's concurrency, and about the trade-offs related to achieving the best choice in terms of performance, resource usage, and development effort?
- What are the trade-offs between performance and programmer effort (which can be expected to be reduced) that can be achieved by using the high-level approaches of SDSoC OpenCL and SDSoC C++ compared to using the lower abstraction level of the Vivado HLS?
- Is it feasible to consolidate a methodology for mapping the concurrency in weather and climate applications to FPGAs to improve the FPGA programmability for traditional HPC software programmers?

1.3 Thesis Contributions

The thesis presents the following contributions, from the perspective of informing traditional HPC software programmers as they consider adopting FPGA technologies:

- An exploratory study of the use of HLS mechanisms with SDSoC OpenCL and Vivado HLS for mapping the concurrency in a single *SWM* application (targeting instruction-level-, data- and functional-parallelism) and multiple *SWM* kernels to a Xilinx Ultrascale+ FPGA.

- An exploratory study aiming to investigate the extent to which it is possible to achieve replication of an existing FPGA design for the *MatVec* kernel created using (lower-level) Xilinx High-Level Productivity Design Methodology Vivado approach in the (higher-level) HLS tools SDSoC OpenCL and SDSoC C++.
- A comparison study and analysis of the performance, resource usage and programmability issues between the SDSoC and Vivado HLS SWM single and multi-kernel implementations.
- A comparison study of Vivado HLS and SDSoC OpenCL and SDSoC C++ *MatVec* implementations and an analysis of the techniques available to be used in the approaches with a focus on the differences in the three approaches which result in their performance, scalability and resource usage and a discussion of programmability issues.

1.4 Thesis structure

The structure of the thesis is as follows: Chapter 2 presents the necessary related background and the related work. Here, the FPGA is explained as a technology. The three target HLS tool-sets (SDSoC OpenCL, SDSoC C++ and Vivado HLS) methodologies and design flows are described. Moreover, the different HLS mechanisms available in each HLS tool-set for mapping the concurrency in HPC applications and kernels are explained. The two weather and climate example applications used, SWM and LFRic, are then described, along with a discussion of the concurrency available for exploitation in parallel implementations. Chapter 3 establishes the methodology for conducting the two exploratory and two comparison studies presented in this thesis. First, the target Xilinx ZynQ UltraScale+ platform is described. Following this, the methods and techniques used in the exploration studies are presented and, finally, the method and metrics used in the comparison studies are detailed. The first exploratory study is presented in two chapters. Chapter 4 presents the first part of the first exploratory study which explores the concurrency level mapping strategies available within a single SWM kernel (L100) using the SDSoC OpenCL and the Vivado HLS approaches. Chapter 5 presents the second part of the first exploratory study which explores the mapping of the SWM application's multiple kernels using the SDSoC OpenCL and the Vivado HLS approaches. Chapter 6 compares the concurrency mapping solutions

developed using the SDSoC OpenCL and the Vivado for mapping the SWM application's single and multiple kernels. Chapter 7 establishes the exploratory study for the LFRic implementation using the SDSoC OpenCL, SDSoC C++ and Vivado HLS tools and Chapter 8 presents and analyses the results of the comparison study of the LFRic implementations developed with each of the three HLS tools (SDSoC OpenCL, SDSoC C++ and Vivado). At the end of Chapter 7 a summary discussion of the findings from the two exploratory studies is presented, and at the end of Chapter 8 we present a summary discussion of the findings from across the two comparative studies. Finally, conclusions, limitations of the research and possible future work are presented in Chapter 9.

1.5 Publications

This thesis research has resulted in the following publications:

- Alghamdi, M., Riley, G. and Ashworth, M. Concurrency Mapping to FPGAs with OpenCL: A Case Study with a Shallow Water Kernel [[ARA21b](#)].
- Alghamdi, M., Riley, G. and Ashworth, M. A Comparison of Vivado HLS, SDSoC C++ and OpenCL for Porting a Matrix-vector-based Climate model mini-app to FPGAs [[ARA21a](#)].
- Alghamdi, M., Riley, G. Design Space Exploration of Concurrency Mapping to FPGAs with OpenCL: A Case Study with Shallow Water Model Kernel [[Alg20](#)].

Chapter 2

Background and Related Work

This chapter establishes the necessary background for this PhD research. It starts with introducing FPGAs and High-level synthesis tools, focusing on the background details (methodologies and design flow) of the target HLS tools used in this research (SDSoC OpenCL SDSoC C++ and Vivado HLS). This chapter also presents the optimisation methods that are relevant to the research study implementations and related to the OpenCL programming language and the target HLS tools. The two HPC applications, the shallow water model and the LFRic, used in this PhD research study are also explained and discussed in this chapter. The final section in this chapter presents the related literature work and discusses where this PhD contribution lies between those studies.

2.1 Field Programmable Gate Arrays (FPGA)

The Field Programmable Gate Array (FPGA) is a programmable device. Unlike other processing devices, such as GPUs and CPUs, which have pre-defined architectures, FPGAs enable the logic architecture to be fully customizable, whereby the programmer can tailor the hardware solution to the application needs [WHU18]. FPGAs are based on three building blocks that are configured to implement a specific application. These components are a matrix of configurable logic blocks (CLBs), input and output blocks, and communication resources. The main element of an FPGA is the matrix of CLBs, which comprises thousands of logic blocks of different types [KHZ16] such as:

- **Look-Up Tables (LUTs)** which performs logic operations.
- **Multiplexers and Flip-Flops (FF)** which store the results of LUTs.

There are also other components including these:

- **Digital Signal Processors (DSPs)** which are embedded arithmetic logic units (ALU).
- **Block RAM** which is on-chip dual-port RAM modules that can provide storage for a relatively large set of data. The ports can both be used for reading and writing or one for read and another for writing.

In the past, FPGAs were considered low density, low volume ASIC replacements; however, following Moore's law, they became faster and denser. Several kinds of FPGA-based system types are accessible today. They range from heterogeneous systems that couple FPGAs with conventional CPUs through PCIe such as [Xil] to system-on-chip (SOC) FPGAs that mix ARM processors with programmable logic on the same fabric [Wine, Int]. Compared to CPUs and GPUs, FPGAs typically run at an order of magnitude lower clock frequency. In several workloads (especially floating-point-based ones), GPU performance is either very close or slightly better than an FPGA; However, both CPUs and GPUs power efficiency (performance per watt) lag significantly behind FPGAs, as shown in recent studies [BNM⁺20, XX20, NWS⁺20, JF20, GLR19, AMI21, ZP20, QDL⁺19, CHPB21, Zoh18].

FPGAs are gaining popularity in both industrial and academic research as a way of implementing application-specific accelerators [KHZ16]. Until recently, most FPGA users would have been hardware designers with extensive experience and knowledge of circuit design using conventional hardware description languages (HDL). However, a wide range of High-Level Synthesis (HLS) FPGA programming languages and supporting tools have been developed to improve FPGA usability.

2.2 High Level Synthesis Tools

HLS tools are a step that the commercial and academic communities have deployed to facilitate FPGAs programmability. HLS tools have improved the software development productivity of using FPGAs by automating the design creation process from the algorithm level to RTL. They generate the RTL design from an algorithm written in a high programming language such as C, C++, OpenCL or Java [KHZ16]. HLS tools hide several traditionally low-level manual design tasks from the programmers, such as: resource allocation, for example determining the type and amount memory

elements to use and the types of associated operators; scheduling, e.g., assigning the algorithm's operations to time slots (clock cycle); resource binding, such as assigning the algorithm's operations to specific operators and memory elements. In addition, the HLS tools typically automate the interface synthesis, such as the interface type generation (i.e., data or control signals) between the generated design and peripherals such as the memory interface.

HLS tools offer several advantages to the FPGA programmer. For example, reducing the coding time dramatically compared to the use of low-level descriptive languages (such as Verilog and VHDL) [LY16], which results in both development time savings and fewer design mistakes. Design optimisation in HLS tools is achieved through tweaking the source code and tool options, which lead to extensive design space exploration opportunities, an issue explored in this thesis. Design verification time is reduced with the use of HLS tools due to the ability of most of the tools to generate test benches and setting the data test to validate the source code.

Table 2.1 presents a summary of the most common HLS tools that are available, either academic or commercial. The following subsections will focus on the SDSoC OpenCL, SDSoC C++¹ and Vivado HLS tools which are used in this thesis.

2.3 SDSoC HLS

SDSoC is a development environment that provides an easy to use Eclipse-based IDE for C/C++ and OpenCL application development. SDSoC provides a high-level FPGA programming model by combining the processing system, accelerators, data movers, signalling and drivers under one infrastructure. This abstraction enables shorter FPGA development time and simplifies the developer's view of the interface between the software and hardware. This environment contains two FPGA high-level compilers: `sdscc/sds++` for C/C++ kernels and `xocc` for OpenCL [Xil21h]. Those compilers invoke the Vivado HLS tool in order to compile the C/C++ and OpenCL functions into a bitstream to load onto the programmable logic. Figure 2.1 shows the SDSoC environment top-level user design flow. The first step is profiling the application to identify the compute-intensive candidate portion(s) of the application for acceleration. Following

¹Xilinx Vitis HLS tool is a rebranding of SDSoC OpenCL and SDSoC C++, and this research and conclusions are applicable to the Vitis too.

#Number	Tool Name	Availability	Computation type	Input language	Target Architecture	Year
1	Vivado Hls	Commercial	DataFlow & ControlFlow	C/C++,SystemC	Specific FPGA Board	2013
2	SDSoC	Commercial	DataFlow & ControlFlow	OpenCL/C/C++	Specific FPGA Board	2015
3	SDAccel	Commercial	DataFlow & ControlFlow	OpenCL/C/C++	Specific FPGA Board	2015
4	MaxCompiler	Commercial	DataFlow	MaxJ	Specific FPGA Board	2010
5	OmpSS	Commercial	ControlFlow	OpenMP	Specific FPGA Board	2016
6	Altera SDK/OpenCL	Commercial	DataFlow & ControlFlow	OpenCL/C	Specific FPGA Board	2013
7	Bluspec	Commercial	DataFlow & ControlFlow	BSV	FPGA	2007
8	Catapult	Commercial	DataFlow & ControlFlow	ANSI C++, SystemC	ASIC,FPGA	2004
9	CHC Compiler	Commercial	ControlFlow	Standard C	FPGA	2008
10	C-to-Silicone	Commercial	DataFlow & ControlFlow	C,C++,SystemC	ASIC,FPGA	2008
11	CyberWorkBench	Commercial	DataFlow & ControlFlow	BDL	ASIC,FPGA	2011
12	Cynthesizer	Commercial	DataFlow & ControlFlow	SystemC,C	ASIC,FPGA	2004
13	GAUT	Academic	DataFlow & ControlFlow	C	ASIC,FPGA	2010
14	HDL Coder	Commercial	ControlFlow	Matlab, Simulink	ASIC,FPGA	2015
15	HIPAcc	Commercial	DataFlow	C++ Embedded DSL	FPGA	2014
16	Impulse C	Commercial	DataFlow & ControlFlow	ANSI C	FPGA	2003
17	LabVIEW FPGA	Commercial	DataFlow & ControlFlow	G	Specific FPGA Board	-
18	LegUp	Academic	DataFlow & ControlFlow	ANSI C	FPGA	2011
19	Merlin Compiler	Commercial	DataFlow	C /C++	FPGA	-
20	PARO	Academic	DataFlow	PAILA	FPGA	-
21	ROCCC	Academic	ControlFlow	C	Specific FPGA Board	-
22	SPIRAL	Both	DataFlow	SPL	ASIC,FPGA	-
23	Trident	Academic	ControlFlow	C subset	FPGA	-
24	Synphony C Comp.	Commercial	DataFlow & ControlFlow	C/C++	ASIC,FPGA	2010
25	eXCite	Commercial	-	C	-	2001
26	CoDeve-loper	Commercial	-	Impulse-C	-	2003
27	CtoS	Commercial	-	SystemC / TLM/ C++	-	2008
28	DK Design Suite	Commercial	-	Handel-C	-	2009
29	Bambu	Academic	-	C/C++	FPGA	2001
30	DWARV	Academic	-	C subset	FPGA	2012
31	triSYCL	Commercial	-	OpenCL	FPGA	-
31	Intel® oneAPI	Commercial	-	DPC++ Language	Specific FPGA board	2018
32	Vitis	Commercial	DataFlow & ControlFlow	OpenCL/C/C++	Specific FPGA Board	2019

Table 2.1: Summary of the current available HLS tools and their properties: **Availability** Some of the tools are made freely accessible by the developers, while others require a licence. **Target architectures** There are two main FPGA board producers, Xilinx [Wind] and Altera [Wina]; Some target a specific FPGA board; while others target both FPGA platforms. **Computation type** demonstrates whether a specific tool supports the dataFlow or controlFlow paradigm [SMB⁺]. **Input Language** describes the design entry language of the tool. [KHZ16] [NSP⁺16]. Information in the empty cells is not available.

that, the SDSoC system compiler can be invoked to generate a complete system-on-chip and SD-card boot image for the application. The application code can be instrumented to analyze performance and help optimize the hardware functions using a set

of directives within the SDSoC environment. After the bitstream generation, the tool generates estimation reports to support further optimizations and other information. The programmer’s interaction with the FPGA system in the SDSoC environment is greatly simplified since the SDSoC compilers abstract the FPGA design flow. The SDSoC compilers automatically choose the system design, such as the data transfer ports to use and the data mover IP blocks, etc [Xil21h].

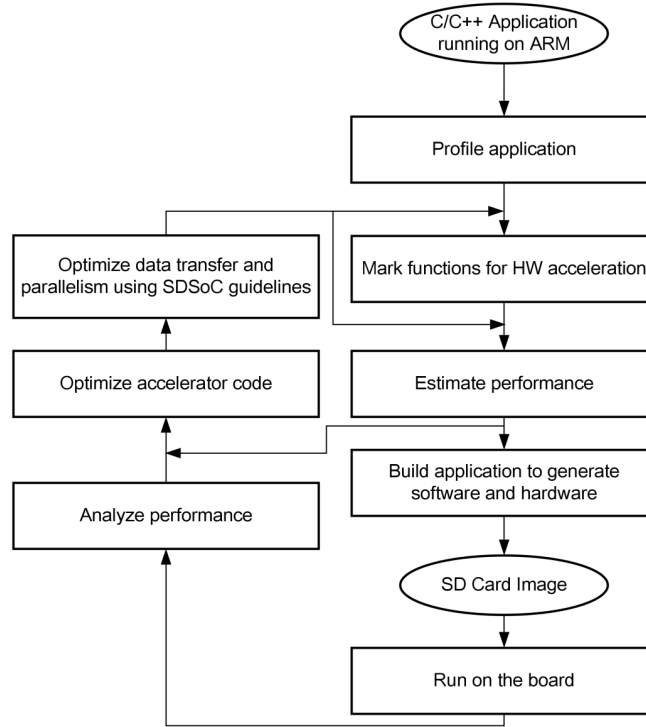


Figure 2.1: SDSoC Design Flow [SG20].

The following subsections present the OpenCL and C++ programming languages within the SDSoC environment.

2.3.1 SDSoC OpenCL

OpenCL is a parallel programming standard that is developed by the Khronos Group for addressing the challenges of programming heterogeneous compute platforms and multi-core systems. It is a programming model that is supported by different hardware platforms, which provides for the utilisation of devices from multiple vendors [Sca11].

OpenCL supports the development of functional and portable “close-to-the-metal”

software by providing a programming language and a runtime API [Winb]. In addition, a set of low-level hardware abstractions such as platform, memory and execution model for exposing the underlying hardware details is provided. Understanding the translation of these OpenCL concepts into the physical implementations on the FPGAs is an important step for achieving efficient implementation.

The **OpenCL platform** defines all the available hardware that is capable of executing an OpenCL program. The available host and one or more OpenCL compute devices are grouped when the OpenCL platform is defined. In the SOC system used in this thesis, the host is an ARM CPU responsible for the general OpenCL tasks such as the launch duties of the OpenCL applications. The device is the FPGA hardware implementation on which the OpenCL application's compute kernels are executed.

The **OpenCL execution model** defines a kernel's execution. There are two types of kernels, one is called *Task* kernel and the other is *NDRange* kernel. The OpenCL execution model with task kernels executes the kernel as a single work item while, an NDRange kernel is executed within the concept of an index space. For instance, an index space which is easy to understand is that of the C/C++ `for` loop. Index spaces in OpenCL are called NDRange, and they can have 1,2 or 3-dimensions [MGMG11].

The **OpenCL Memory model** defines the memory hierarchy and behaviour of the memories that the OpenCL applications can use. This memory hierarchy representation is common across all OpenCL implementations. However, it depends on the individual vendors mapping definition of the OpenCL memory model to specific hardware. For example, Figure 2.2 shows the representation of the OpenCL memory model in Xilinx SOC Ultrascale+ FPGAs that we target in this thesis.

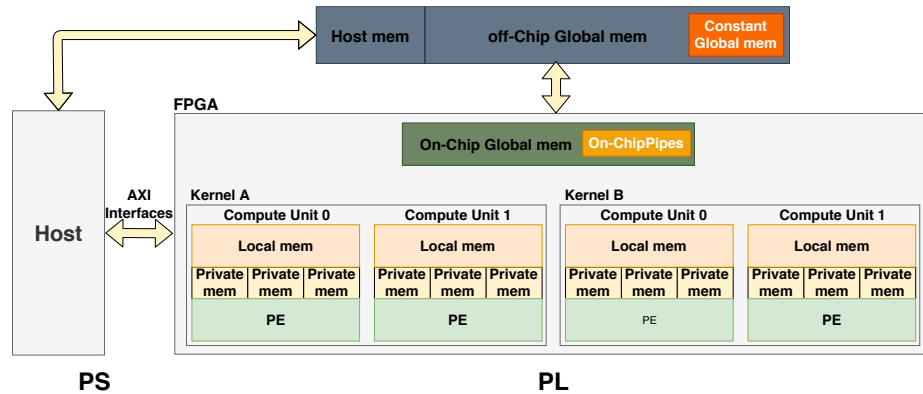


Figure 2.2: OpenCL memory model in Xilinx Zynq UltraScale+ MPSoCZCU102.

The OpenCL memory model is translated onto the FPGAs as follows:

- **Host memory** represented by a memory address space within an off-chip-memory (e.g. DDR) that resides outside the FPGA's fabric area. This memory is a shared memory between the processing system (PS) and programmable logic (PL) parts of the FPGA. The OpenCL buffer allocation and de-allocation is the responsibility of the host and is controlled by a handshake method between host and device that switches the data access rights to memory buffers between the host and the device. The data is transferred from the host address space to the other device spaces using the OpenCL API when the device kernels require the data.
- **OpenCL global memory** can be either represented by the shared off-chip memory or use distributed memories (e.g. BlockRAM) that reside within the FPGA's fabric area.
- **Constant Global Memory** is a system memory region that has full rights access from the host and read only access rights for the OpenCL device. This memory model is implemented within the off-chip memory, as shown in figure 2.2.
- **Local Memory** is on-chip memory that is accessible only within one compute unit. This memory type, which is typically implemented using registers or BlockRAMs, can be implemented either as RAM or ROM, covering on-chip global, local, and private memory types.
- **private memory** is a memory that is only accessible to an individual work-item. It is implemented using registers or BlockRAMs in the FPGA fabric.

2.3.2 SDSoC C++

The C++ design approach consist of three coding phases: a standard C++ host application that runs on a CPU; the C++ functions that are selected to be accelerated by the FPGA, which are handled by the SDSoC *sds++* compiler; and the design of the data motion network that manages the data movement between the CPU and the FPGA hardware.

The host C++ application allocates the data buffers in the DDR memory shared with the FPGA hardware. The `sds_alloc` function is recommended [SG20] for the memory allocation because the data will then be stored in physically contiguous memory. The data movement in the C++ approach is specified through the data motion network [Xil21h] that manages the data movements. A data motion network has three components that programmers can control with *pragmas* for a better choice that suits

the target kernel design. The most critical components are the *data mover engines* which are FPGA IP blocks for transferring the data between the CPU and the FPGA accelerator.

Figure 2.3 shows a highlight of the three data motion network components available, which are labelled as A, B and C. The three components are: (A) *System port*



Figure 2.3: “SDSoC data motion network components. PS means the processing system CPU part. Acc means the accelerator par which is the FPGA (A) is the system port type. (B) are the data mover engines. (C) is the accelerator interface port type” [Xil21h].

which is the port that connects the data mover engines to the CPU, (B) *Data mover engine* which is an FPGA IP block for transferring the data between the CPU and the FPGA accelerator and (C) *Accelerator Interface port* which is the accelerator port for transferring the data between the data mover and the accelerator [Xil21h]. The choice of component hardware affects system performance and efficiency. Therefore, finding the best choices for the kernel design is vital for performance.

The SDSoC *sds++* compiler can analyse the design and choose the three components automatically. However, in this HLS approach, the designer has the freedom to override the compiler choices. The choice of the system and accelerator interface ports always depends on the data mover engine selected [Xil21h].

System port is the port connection type that connects the data mover and the host side. For example, the Zynq® UltraScale+™ MPSoC board provides the following System port options:

- High performance ports *ACP* and *AFI*. *ACP* is a cache-coherent port and *AFI* is a non-cache-coherent port.
- PL-based DDR memory controller port *MIG*
- Stream port

The SDSoC *sds++* compiler analyze the choice of memory attributes based on the data transferred and data motion type to identify the appropriate system port. However, using the following *pragma*, `#pragma SDS data sys_port(arg:port)`, the programmer can override the compiler decision.

Table 2.2: The properties details of the available Data movers engines in the SDSoC C/C++ approach.

Data Mover Engine	Physical Memory Contiguity	Data Size (bytes)
AXI_LITE	No	-
AXI_DMA_SIMPLE	Contiguous	<8 M
AXI_DMA_SG	No	>300
AXI_FIFO	No	<300
Zero-Copy	Contiguous	-

Data mover engine is an FPGA IP block that is responsible for transferring the data between the host-side and the accelerators and among accelerators. There are five data mover engines options available. The engine choice depends on the properties and size of the data being transferred. In addition, the selection of the data mover engines is a trade-off between performance and resource usage. Table 2.2 shows the five data mover engines and their properties. Engine *AXI_LITE* is suitable only for transferring scalar data, while the other engines are for transferring arrays. *AXI_DMA_SIMPLE* is the faster transfer engine and only supports up to 8 MB of data. The data must be allocated contiguously in the DDR memory when using this data mover engine. *AXI_DMA_SG* is a scatter and gather engine that is slow and consumes high resources, but it has fewer limitations compared to the other options. This engine accepts transferring data that are not physically contiguous. *AXI_FIFO* is a data mover engine that does not require many hardware resources compared to the other engines. In addition, it is a slow engine and limited to transferring less than or equal to 300 bytes of data. *Zero-Copy* is a unique data mover engine because it covers the choice of the accelerator interface and the data mover. The use of the *AXI_LITE*, *AXI_DMA_SIMPLE* and *AXI_DMA_SG* engines require an explicit data copy from the host to the accelerator via the data mover. However, in the *Zero-Copy* case, the SDSoC compiler generates an AXI-Master accelerator interface that fetches the data from the host as specified in the accelerator code. The use of this engine requires physically contiguous memory data allocation.

The SDSoC C++ compiler analyzes the transferred array data in terms of memory contiguity and size and selects the appropriate data mover. However, in some cases this analysis is not possible. In this case SDSoC provides the programmer with SDS pragmas for specifying the data mover and for specifying the memory attributes and size. This SDS pragma: `#pragma SDS data data_mover (A:<data_mover name> ,`

B:<*data_mover name*>) is used to define the use of either *AXI_LITE*, *AXI_DMA_SIMPLE* or *AXI_DMA_SG* engines. For choosing the *Zero Copy* engine the programmer can use this pragma: `#pragma SDS data zero_copy(arg[offset:size])`.

If the SDSoc C++ compiler cannot analyse the memory attribute of the transferred data, it will issue a warning message asking the programmer to specify the memory attributes and size. The following two pragmas can be used by the programmer to specify memory attributes: `#pragma SDS data mem_attribute(arg:contiguity)` and data size `#pragma SDS data copy(arg[offset:size])`.

Accelerator Interface port defines the connection between the data mover and the hardware accelerator. Register interface port is used if the transferred data is a scalar. However, in the case of array data, there are two port types options: *RAM interface* or *a streaming interface*. The RAM interface is used for random data access and requires transferring the whole array of data before access can occur. While in the streaming interface option, the data are accessed sequentially and it does not require the whole array being transferred. This port interface allows the pipelineing of the array elements for processing. `#pragma SDS data access_pattern(arg:pattern)` is an SDS pragma that the programmer can use to define the accelerator Interface port type. The pattern can be either *RANDOM* for the RAM interface or *SEQUENTIAL* for the streaming interface.

2.4 Xilinx Vivado

Xilinx Vivado HLS tool is a compilation system which analyses a restricted form of C code and schedules operations on the FPGA hardware during a later *synthesis* phase. The design flow methodology in this approach consists of the following three design stages [Xil21d].

Vivado HLS tool (Compile Stage) is the **first stage** in the Xilinx Vivado approach. This stage aims to synthesize a C function code into an IP block. Vivado HLS writes Register Transfer Level (RTL) code which forms the basis of an IP (*Intellectual Property*) Block that can be saved to an IP Repository for later inclusion in a complete FPGA system design. In addition, this tool provides compiler-automated optimizations that are supplemented by programmer-supplied HLS pragmas that invoke and guide a range of optimizations such as pipelining and unrolling, and managing data placement and streaming. In this stage, the programmer focuses on creating an efficient IP block for the HPC workload. The tool provides performance estimation reports that enable

the programmer to improve the kernel code design. Several HLS optimization pragmas are available to be utilized to improve the kernel computational performance, as discussed in Section 2.5. Following this, the final design can be synthesized into a hardware IP block.

Vivado design suite (Link Stage) is the **second stage**. The design methodology in this stage requires the construction of the full system design *manually* using the Vivado Design Suite. That involves specifying plenty of low-level hardware details to configure the final generated system design and gives the application developer considerable fine control. The HLS-generated application-specific IP blocks are manually integrated with IP from the Vivado IP Catalog and, potentially, from third-party sources. The generated IP block for the target kernel from stage one can be seen from the Vivado IP Catalog in the Vivado design suite. For example, see Figure 2.4 which shows an example of an IP Repository created using the Vivado HLS (as in stage one) in the Vivado IP Catalog. The programmer can integrate the IP block to a full system design before generating the final FPGA bitstream file. Figure 2.5 shows an example of a full Vivado system design with the IP kernel from Figure 2.4 integrated into the design. The programmer in this stage is responsible for choosing the appropriate IP blocks in order to support the communication between the kernel IP and the Zynq (ARM host) IP block. In addition, the responsibility involves the choice of the connection ports, setting the data path sizes and choosing and creating the memory solution. The programmer manually performs the address management and the setting of the frequency of clocks. The Vivado environment adjusts the IP blocks address automatically, see Figure 2.6. However, the programmer can override those addresses manually, if needed.

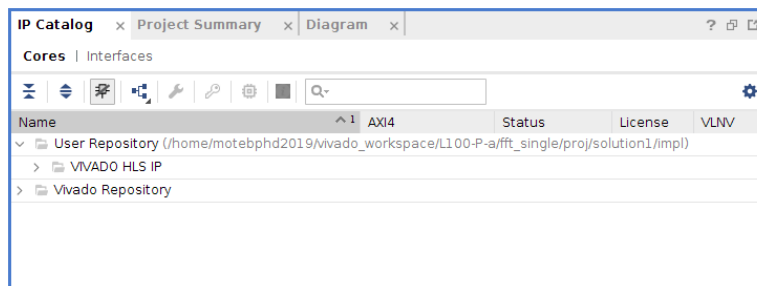


Figure 2.4: An example of IP Repository in the Vivado IP Catalog.

The **third stage** is creating the **ARM CPU Host** application. The host application code manages the data transfer of input and output data between the host and the HLS kernel, and the host programmer has to manually configure the data addresses. The

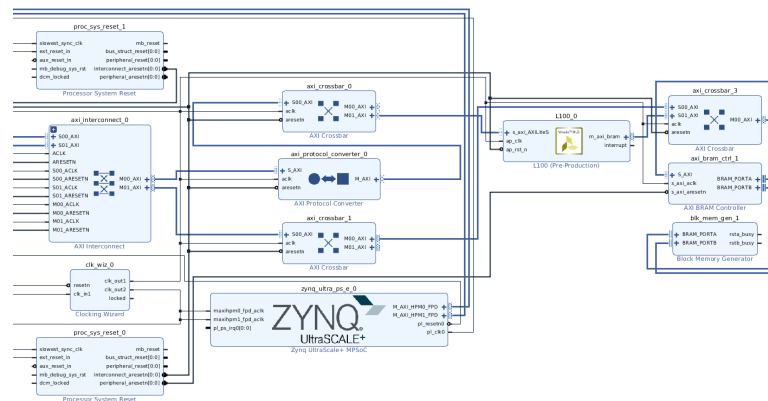


Figure 2.5: A full Vivado system design example with an IP kernel integrated to the design.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
zynq_ultra_ps_e_0					
Data (40 address bits: 0x00A0000000 [256M] , 0x0400000000 [4G] , 0x1000000000 [224G] , 0x0500000000 [256M] , 0x0500000000 [4G] , 0x4800000000 [224G])					
l100_0					
axi_bram_ctrl_1	S_AXI	Mem0	0x00_A000_0000	1M	0x00_B00F_FFFF
l100_0					
Data_m_axi_bram (32 address bits: 4G)					
axi_bram_ctrl_1	S_AXI	Mem0	0x8000_0000	1M	0x800F_FFFF

Figure 2.6: An example of the Vivado address editor.

host program also manages the invocation of kernels. At the end of the three stages, the application is ready for execution on the host and FPGA. The following points summarise the steps in managing the Vivado host code:

- Open device tree (/ui0, /ui01). See line 2 in Listing 2.1.
- Map device tree to Zynq ports (i.e. HPM0, HPM1) using open and mmap system calls, to control the kernel IP blocks. See lines 4-9 in Listing 2.1.
- Manually configure the IP blocks and memory solution addresses. See lines 11-16 in Listing 2.1.
- Control Kernels IP blocks execution through the access to the IP blocks' control registers (i.e. configure AP_START bit to 1 to start IP block execution and AP_IDLE for polling the IP block status). See lines 18-28 in Listing 2.1.

Listing 2.1: Vivado ARM CPU code example

```

1 // dtc command
2 dtc -I fs -O dts /sys/firmware/devicetree/base
3

```

```

4      // open and mmap system calls
5      fd = open(device , O_RDWR);
6      fpgamemsize[idev] = 0x0800000;
7      fpgamemory[idev] = (char *)mmap
8      (NULL,fpgamemsize[idev],PROT_READ|
9      PROT_WRITE,MAP_SHARED,fd ,0);}
10
11     //Set up IP block addresses
12     ierr = fpga_add_block(0, 0xA0000000);
13
14     // Set memory solution addresses
15     // (Example BRAM block)
16     ierr = fpga_add_bram(1, 0xB0000000);
17
18     // IP Block start
19     int fpga_start (int ib_block) {
20         *control[ib_block] = 1;
21         return 0;
22     }
23
24     // IP Block status check
25     hold=0;
26     while((*control[ib_block]&4) == 0) {
27         hold++;
28     }

```

2.5 Vivado and SDSoC OpenCL/C++ Optimisation Strategies

As evident in the previous sections, the Vivado, SDSoC OpenCL and SDSoC C++ design flow methodologies are different; however, when it comes to the kernel computational design, they provide the programmer with annotations that can be inserted into the kernel code to guide the HLS compiler for optimizing the kernel computational performance. These annotations are called *HLS pragmas* in the Vivado and the

SDSoC C++ approaches and *attributes* in the SDSoC OpenCL approach. The HLS compilers are designed to analyze the kernel code and automatically apply some annotation optimizations such as `pipeline` and `unroll` when possible. However, auto-compiler kernel optimizations are often not efficient enough to extract the desired performance [HLC⁺13]. The following points present the HLS optimization annotations available in the Vivado, SDSoC OpenCL and the SDSoC C++ relevant to this thesis.

In SDSoC OpenCL there are two levels of optimization mechanisms, one is the optimizations methods related to the OpenCL programming language, and the other is the computational and data movement optimization attributes. The **OpenCL programming language** has three important mechanisms which are: the OpenCL kernel *type*, the OpenCL *number of kernels* and the OpenCL API *command queue(s)*.

An OpenCL kernel for an FPGA can be either a *task* kernel or an *NDRange* kernel [XO20]. *Task kernel* (or Single work-item kernel): Task kernel refers to the execution of the kernel with a single work-group (WG) that contains only one work-item (WI). *NDRange Kernel*: This kernel type exploits data parallelism by processing the kernel data using multiple work-items (WIs). NDRange organizes the WIs in WGs. WGs can be executed simultaneously, and the WIs within each WG can also be processed in parallel [XO20]. The kernel type is specified through the setting of the work-group size of the OpenCL kernel. The following OpenCL attribute is used to specify the kernel's work-group size: `__kernel __attribute__((reqd_work_group_size(N, M, L)))`. The values of *N* define the type of the kernel for the SDSoC OpenCL compiler. In the case of the *task* kernel, *N*, *M* and *L* values should be *1, 1, 1*; while in the *NDRange* kernel those values define one-dimensional, two-dimensional, and three-dimensional NDRanges (and hence work-groups).

OpenCL *number of kernels* results from the software design, reflecting the developer's choice as to how to implement the algorithmic operations of the application in one or more kernels [XO20]. The execution of multiple kernels can be controlled through the choice of the OpenCL API *command queue(s)* selected. Individual command queues can execute kernels *in-order* or *out-of-order* and multiple queues may be used. OpenCL has (host) mechanisms to synchronize the execution of individual kernels through the use of *barriers* and *events* [XO20].

2.5.1 Kernel Computation and Data-movement Optimisation Attributes and Pragmas

This subsection presents in Table 2.3 the list of optimization annotations that are relevant to this thesis and which are available in the SDSoC OpenCL/C++ and Vivado HLS tools. Table 2.3 shows the optimization methods in the two computational and data-movement categories and introduces the annotations (attribute or pragma) associated with each method.

Table 2.3: SDSoC OpenCL/C++ and Vivado HLS Kernel Computation and data-movement Optimisations Attribute and Pragmas.

Optimisation method	Computation Optimisation	Data movement Optimisation	OpenCL Attributes	C++ and Vivado Pragmas
Pipelining Loops	Yes	-	<code>__attribute__((xcl_pipeline_loop))</code>	<code>#pragma HLS pipeline II = <int></code>
Pipelining Work Items	Yes	-	<code>__attribute__((xcl_pipeline_workitems))</code>	-
Unrolling Loops	Yes	-	<code>__attribute__((opencl_unroll_hint(n)))</code>	<code>#pragma HLS unroll factor = <N></code>
Dataflow	Yes	-	<code>__attribute__((xcl_dataflow))</code>	<code>#pragma HLS dataflow</code>
Multiple Compute Units	Yes	-	Multi-kernels and GUI feature	Multi-Kernels
Pipes	-	Yes	<code>pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)))</code>	-
Data Streaming	-	Yes	-	<code>#pragma HLS stream variable = <variable> depth = <int> dim = <int> off</code>
Burst Mode	-	Yes	<code>__attribute__((xcl_pipeline_loop))</code>	<code>#pragma HLS pipeline II = <int></code> OR C memcpy function
Max Memory Ports	-	Yes	GUI feature	-
Data Array Partitioning	-	Yes	<code>__attribute__((xcl_array_partition(<type>, <factor>, <dimension>))</code>	<code>#pragma HLS array_partition variable = <name>\<type> factor = <int> dim = <int></code>

The kernel **computational optimisation methods** are the following:

The *pipeline* attribute (`__attribute__((xcl_pipeline_loop))`) and `#pragma HLS pipeline II = <int>` pragma is used to maximize the calculation throughput and improve the latency of the kernel by keeping several stages of the kernel hardware elements implementing an algorithms step busy. The most important parameter that the pipeline attribute can influence is the *initiation interval* (II) which is the number of

clock cycles before the next iteration can start [OG20]. The smaller the II, the better the performance. For OpenCL NDRange kernels Xilinx provides a specific pipeline attribute (`__attribute__((xcl_pipeline_workitems))`) to pipeline the WIs execution [OG20].

The *loop unrolling* attribute (`__attribute__((opencl_unroll_hint(n)))`) and `#pragma HLS unroll factor=<N>` pragma is used to reduce the number of cycles needed to process a loop's iterations by reducing the loop trip count. Multiple copies of the loop body are created in the FPGA hardware that can be executed in parallel [OG20].

The *Dataflow* attribute (`__attribute__((xcl_dataflow))`) and `#pragma HLS dataflow` pragma is used to exploit the functional parallelism within a single kernel, allowing for parallel execution of the kernel's functions or loops, improving the throughput of the design and decreasing the latency. When the Dataflow method is utilized, individual channels are created to store the results of each Dataflow task. Those channels can be by default either simple FIFOs (for scalar variables) or ping-pong block RAM registers (for variables like arrays). The Dataflow tasks throughput is only limited by the availability of the input and output buffers. If data is accessed in sequential order, the channels are implemented as a FIFO of depth 2; otherwise, they are implemented as two block RAMs, each defined by the maximum size of array data [Xil21d].

The use of *multiple Compute Units, CUs* increases the level of parallelism by utilising more FPGA resources to compute the kernel operations. The kernel operations can be executed by multiple (different) CUs or creating multiple CUs for same the kernel to enhance the spatial parallelism. This is particularly useful, especially with NDRange kernels with multiple WGs where a CU can be created per WG [OG20]. The Xilinx Zynq UltraScale+ MPSoC can accommodate up to sixty Compute Units.

The data-movement optimisations are mainly for improving the data movement between the host and the FPGA kernels and between kernels. The kernel **data-movement optimisation methods** are the following: *Burst Mode*: Data transfer in burst mode is a technique that transfers large volumes of data from the host memory (DDR) to the FPGA kernel local memories (e.g. BRAMs). Instead of issuing multiple single memory transactions. This mechanism ensures the best memory access controller efficiency [SG20]. In the SDSoc OpenCL, Xilinx recommends the use of *pipeline* attribute to implement the *Burst Mode*. Similarly, in the SDSoc C++ and Vivado, *Burst Mode* can be implemented using the *pipeline* attribute over the data read/write loops.

The C `memcpy` function can also be used to implement the *Burst Mode*.

Max Memory Ports: DDR memory has several memory access ports; the Xilinx Zynq UltraScale+ MPSoC board has four ports. Therefore, this optimisation option can increase the number of available DDR memory ports that the CUs in the design can use to access the data from the DDR, either in burst mode or in individual memory accesses per clock cycle [OG20]. In OpenCL, this feature can be enabled through the SDSoC GUI or by setting a compiler flag. In the SDSoC C++, this feature is enabled automatically, while in the Vivado, it is managed manually in the system design blocks.

Array Partitioning: The FPGAs BRAMs have a limited number of data access ports, which can limit the efficiency of the read/write operations. Therefore more than two accesses per cycle will cause conflicts and reduce the Initiation Interval. The use of the *array partitioning* attribute and pragma can improve the BRAM memory bandwidth. This optimisation method divides the data array into smaller arrays implemented in multiple physical BRAMs, increasing the number of memory access ports being used. There are three types of array partitioning: block, cyclic and complete [OG20].

In SDSoC OpenCL *Pipes*: are an OpenCL 2.0 specification that is introduced to stream data between OpenCL kernels inside an FPGA device without using the external global memory. Pipes are implemented in the FPGAs as FIFOs. In the SDSoC OpenCL, pipes have to be defined outside all the kernels functions using the pipes attributes, see Table 2.3. In addition, the depth of the pipes is specified within the attribute definition, and the valid depth values are 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768. The access of the pipes is allowed only from the OpenCL kernel, and they can be only accessed using the OpenCL standard functions: `read_pipe()` and `write_pipe()` for non-blocking mode, and `read_pipe_block()` and `write_pipe_block()` for blocking mode. In addition, a given pipe should only have one producer and consumer in other kernels [OG20].

By default, array variables in FPGAs are implemented as RAM. However, if the array data are accessed or produced sequentially, a more efficient implementation for the array data in the FPGA can be using streaming data. The *Data Streaming* data-movement optimisation method is a similar method to the OpenCL *Pipes* method, but to be utilised in the SDSoC C++ and the Vivado HLS approaches. Using the *Data Streaming* pragma in Table 2.3, the array variables will be implemented using FIFOs instead of RAMs for more efficient communication [OG20].

2.6 HPC-based Benchmark Applications

This section provides background on the two main benchmark applications from the weather and climate domain that are the focus of this thesis. The applications are a **Shallow Water Model** and an **LFRic** mini-app. Details of the applications' equations and algorithms are presented.

2.6.1 Shallow Water Dynamics Model

Shallow Water dynamics can be described by a set of hyperbolic partial differential equations that are widely used to model various kinds of flows such as coastal, river hydrodynamics, and oceanic. This dynamics model describes the fluids flow in regions that have vertical dimensions smaller than the horizontal dimension. The dynamics model can also be used to describe widely used atmospheric flow models such as in numerical weather prediction [Vre94].

The Shallow Water Model (SWM) Equations

The SWM studied in this thesis is based on the shallow water equations presented in Sadourny's original paper in 1975 [Sad75] and are also described in [Pap12] which investigated an OpenCL version of SWM. The SWM model has been widely used as a HPC benchmark. In addition, it represents a number of interesting and generalised facts exhibited by numerous HPC codes. These are the multiple kernels, the data must "flow around" from one iteration to the next, and the application's computation is stencil based and iterative. The latter two facets are extremely common in climate and weather computational codes and represent a broad range of HPC codes. The SWM computes mainly wind velocities in x and y directions, potential pressure, as prognostic variables, and also mass fluxes, potential vorticity and fluid surface height as intermediate variables. The model's equations are expressed as

$$\frac{\partial u}{\partial t} - ZV + \frac{\partial H}{\partial x} = 0 \quad (2.1)$$

$$\frac{\partial v}{\partial t} - ZU + \frac{\partial H}{\partial y} = 0 \quad (2.2)$$

$$\frac{\partial P}{\partial t} + \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} = 0 \quad (2.3)$$

where

- The lowercase u and v are the **wind velocities**.
- P is the **potential pressure**.
- The uppercase U and V are the **mass fluxes**.
- Z is the **potential vorticity**.
- H is the **the fluid surface height**.

U , V , Z and H are further defined by the following equations 2.4, 2.5, 2.6 and 2.7.

$$U = Pu \quad (2.4)$$

$$V = Pv \quad (2.5)$$

$$Z = \frac{(\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y})}{P} \quad (2.6)$$

$$H = P + \frac{1}{2}(u^2 + v^2) \quad (2.7)$$

A two-dimensional finite difference model is used to solve the SWM equations that are updated over time and space. An Arakawa C grid [Cha12, HSS88] is used to calculate the finite-difference model as shown in Figure 2.7. The dimensions of the Arakawa C grid are calculated within a rectangle range of $a \leq x \leq b$ and $c \leq y \leq d$. In addition, with selected M and N integer values defining the resolution of the grid.

The elements of the grid are defined as:

$$x_i = i\Delta x + a, \text{ where } i = 0, \frac{1}{2}, 1, \dots, M+1 \quad (2.8)$$

$$y_j = j\Delta y + b, \text{ where } j = 0, \frac{1}{2}, 1, \dots, N+1 \quad (2.9)$$

SWM Mathematical Representation

Mathematically, the SWM equations solution is evolved using multiple steps. First step initializing the wind velocities u and v and potential pressure P are initialized using a

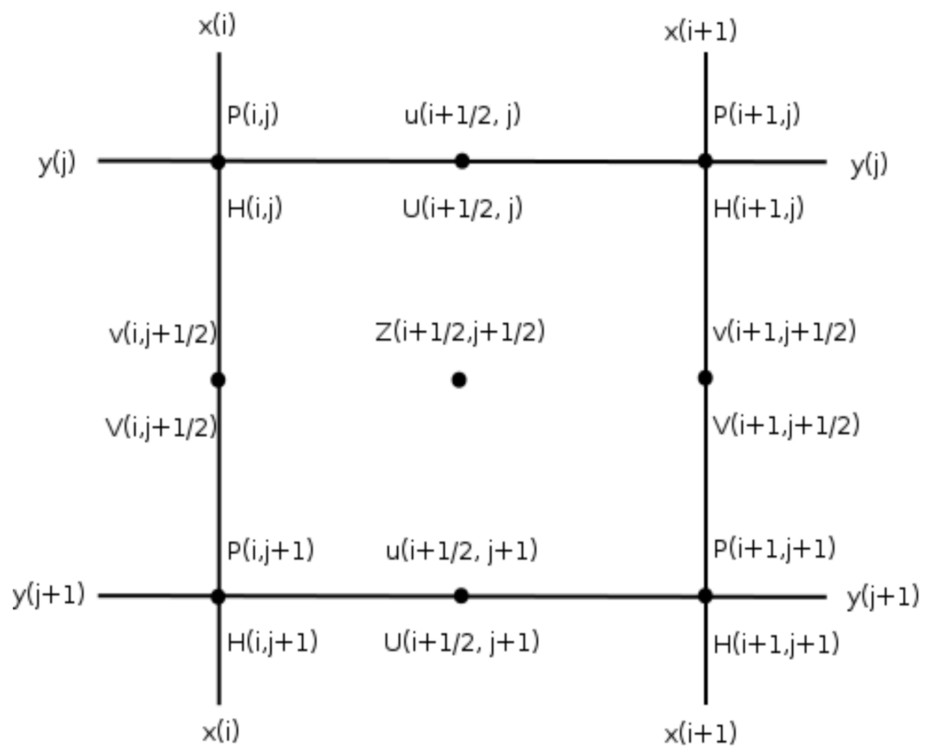


Figure 2.7: An Arakawa-C grid mapping for the shallow water variables [Sad75, Pap12].

stream function, as expressed in equation 2.10.

$$\Psi_{i+\frac{1}{2},j+\frac{1}{2}} = A * \sin(\frac{2\pi x}{b-a}) \sin(\frac{2\pi y}{d-c}) \quad (2.10)$$

Following that, initial velocities are derived as:

$$u_{i+\frac{1}{2},j} = \frac{\Psi_{i+\frac{1}{2},j+\frac{1}{2}} - \Psi_{i+\frac{1}{2},j-\frac{1}{2}}}{\Delta y} \quad (2.11)$$

$$v_{i+\frac{1}{2},j} = \frac{\Psi_{i+\frac{1}{2},j+\frac{1}{2}} - \Psi_{i+\frac{1}{2},j-\frac{1}{2}}}{\Delta x} \quad (2.12)$$

and the potential pressure is initialised as:

$$P_{i,j} = \frac{A^2}{4} [(\frac{2\pi}{d-c})^2 \cos(\frac{4\pi i}{b-a}) + (\frac{2\pi}{b-a})^2 \cos(\frac{4\pi j}{d-c})] + P_0 \quad (2.13)$$

with $P_0 = 50,000$.

Over a large number of repeated cycles, the wind velocities and potential pressure P are calculated using the following equations²:

$$u_{i+\frac{1}{2},j}^{n+1} = u_{i+\frac{1}{2},j}^{n-1} + (Z_{i+\frac{1}{2},j+\frac{1}{2}} + Z_{i+\frac{1}{2},j-\frac{1}{2}}) * (V_{i+1,j+\frac{1}{2}} + V_{i,j+\frac{1}{2}} + V_{i+1,j-\frac{1}{2}} + V_{i,j-\frac{1}{2}}) - (H_{i+1,j} - H_{i,j}) \quad (2.14)$$

$$v_{i,j+\frac{1}{2}}^{n+1} = v_{i,j+\frac{1}{2}}^{n-1} + (Z_{i+\frac{1}{2},j+\frac{1}{2}} + Z_{i-\frac{1}{2},j+\frac{1}{2}}) * (U_{i-\frac{1}{2},j+1} + U_{i-\frac{1}{2},j} + U_{i+\frac{1}{2},j+1} + U_{i+\frac{1}{2},j}) - (H_{i,j+1} - H_{i,j}) \quad (2.15)$$

$$P_{i,j}^{n+1} = P_{i,j}^{n-1} + (U_{i+\frac{1}{2},j} - U_{i-\frac{1}{2},j}) - (V_{i,j+\frac{1}{2}} - V_{i,j-\frac{1}{2}}) \quad (2.16)$$

Next to that, the mass fluxes U and V , the potential vorticity Z and the surface height H are computed using the following discrete equations:

$$U_{i+1/2,j} = \frac{1}{2}(P_{i+1,j} + P_{i,j})u_{i+1/2,j} \quad (2.17)$$

$$V_{i,j+1/2} = \frac{1}{2}(P_{i,j+1} + P_{i,j})v_{i,j+1/2} \quad (2.18)$$

²Superscripts in equations 2.14, 2.15 and 2.16 shows the computation's iteration.

$$Z_{i+1/2,j+1/2} = \frac{\left[\frac{(v_{i+1,j+1/2} - v_{i,j+1/2})}{\Delta x} - \frac{(u_{i+1/2,j+1} - u_{i+1/2,j})}{\Delta y} \right]}{\frac{1}{4}(P_{i,j} + P_{i+1,j+1} + P_{i,j+1})} \quad (2.19)$$

$$H_{i,j} = P_{i,j} + \frac{1}{2} \left[\frac{(u_{i+1/2,j}^2 + u_{i-1/2,j}^2)}{2} + \frac{(v_{i,j+1/2}^2 + v_{i,j}^2)}{2} \right] \quad (2.20)$$

In both of the Arakawa C grid x and y directions, cyclic boundary conditions are applied using **halos**. Therefore, a periodic continuation is applied to updated the elements at the grid boundary, see equations 2.21 and 2.22.

$$f(x+b, y) = f(x+a, y) \quad (2.21)$$

$$f(x, y+d) = f(x, y+c) \quad (2.22)$$

Final step in the mathematical representation of the SWM equations is applying a time smoothing filter by the end of every computation cycle, see equation 2.23.

$$F^{(n)} = f^{(n)} + a(f^{(n+1)} - 2f^{(n)} + F^{(n-1)}) \quad (2.23)$$

SWM Coding Algorithm

The mathematical SWM equations are transformed into a form of OpenCL code that has nine kernels³, as depicted in Figure 2.8 and the source code in appendix A.

The SWM algorithm defines a group of $(M+1) \times (N+1)$ sized double-precision floating-point arrays. Each array stores the data of the dependent variables from the previous equations. The arrays are named, corresponding to the names used in the SWM equations. The the wind velocities and the potential pressure are interpret as (u, v, P) . The mass fluxes in the horizontal and vertical directions, arrays U and V , are interpreted in the code with "C indicating capital then U or V , (CU, CV). The potential vorticity and the surface height are interpret as Z and H .

The calculation of each array is processed in two phases: The *first phase* computes the array's elements values over the $M \times N$ matrix. The *second phase* updates the boundary conditions by copying values to the **halos**. Figure 2.9 shows the halo regions in each array. The halo regions are updated by mirroring border elements of each

³SWM was initially written in Fortran by Paul Swarztrauber for the US National Center for Atmospheric Modelling (NCAR). The supervisor of this thesis, Graham Riley, provided a CUDA OpenCL version.

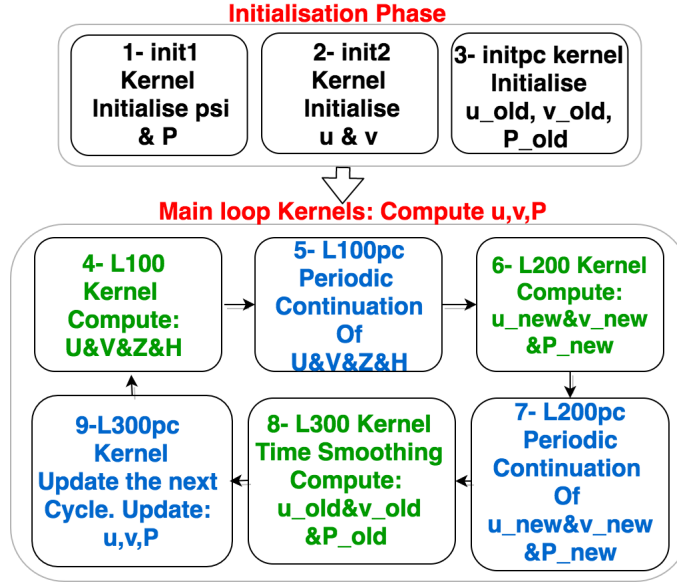


Figure 2.8: SWM nine OpenCL kernels representation.

array, see Figure 2.9. Two "helper" arrays (i.e. u_{new} and u_{old}) are introduced in the code to store the values of the previous and next iteration elements, see Figure 2.8 and the source code in appendix A. This method called for time smoothing to reduce high frequency oscillations in the computed velocities.

The SWM OpenCL code is organized as nine kernels. Figure 2.8 shows a high abstract-level picture of the nine kernels, where each kernel represents a step from the solution of the mathematical equations. Each step is given a kernel name. For example, Kernel 1 (*init1*) initializes an array called psi and P , kernel 2 (*init2*) initializes u and v , and kernel 3 (*initpc*) keeps the old u , v , and P values that are needed in the first iteration of the main loop. In a large number of cycles *L100* kernel computes CU , CV , Z , and H ; then kernel 5 (*L100pc*⁴) updates the boundaries of them. Kernel *L200* computes new data of u , v and P using the old values from kernel 3. Then the boundaries of those values are updated in kernel 7 *l200pc*. Kernel 8 (*l300*) does time smoothing while kernel 9 (*l300pc*⁵) updates u , v , and P for the next cycle. Kernels 4 to 9 are the main kernels computed, typically for 4,000 iterations in our simulation runs. The nine kernels are processed sequentially because they depend on each other; however, several operations can be executed in parallel within each kernel. Figure 2.10 shows a highlight of the data movement between the SWM's five main kernels. This figure also shows the data dependencies between the kernels.

⁴pc stands for periodic continuation.

⁵This kernel called only once inside the main loop.

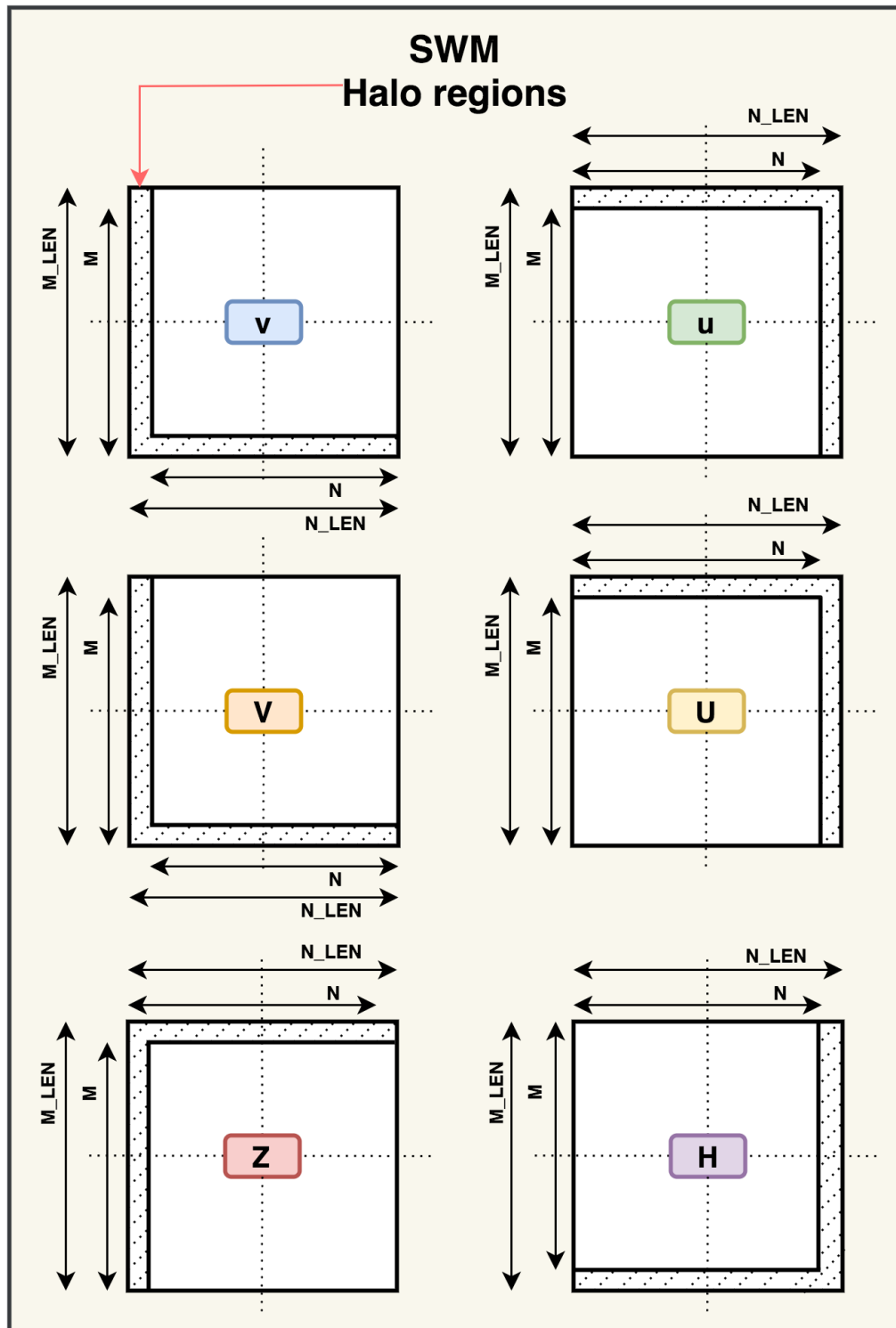


Figure 2.9: Halo regions representation in the SWM arrays elements.

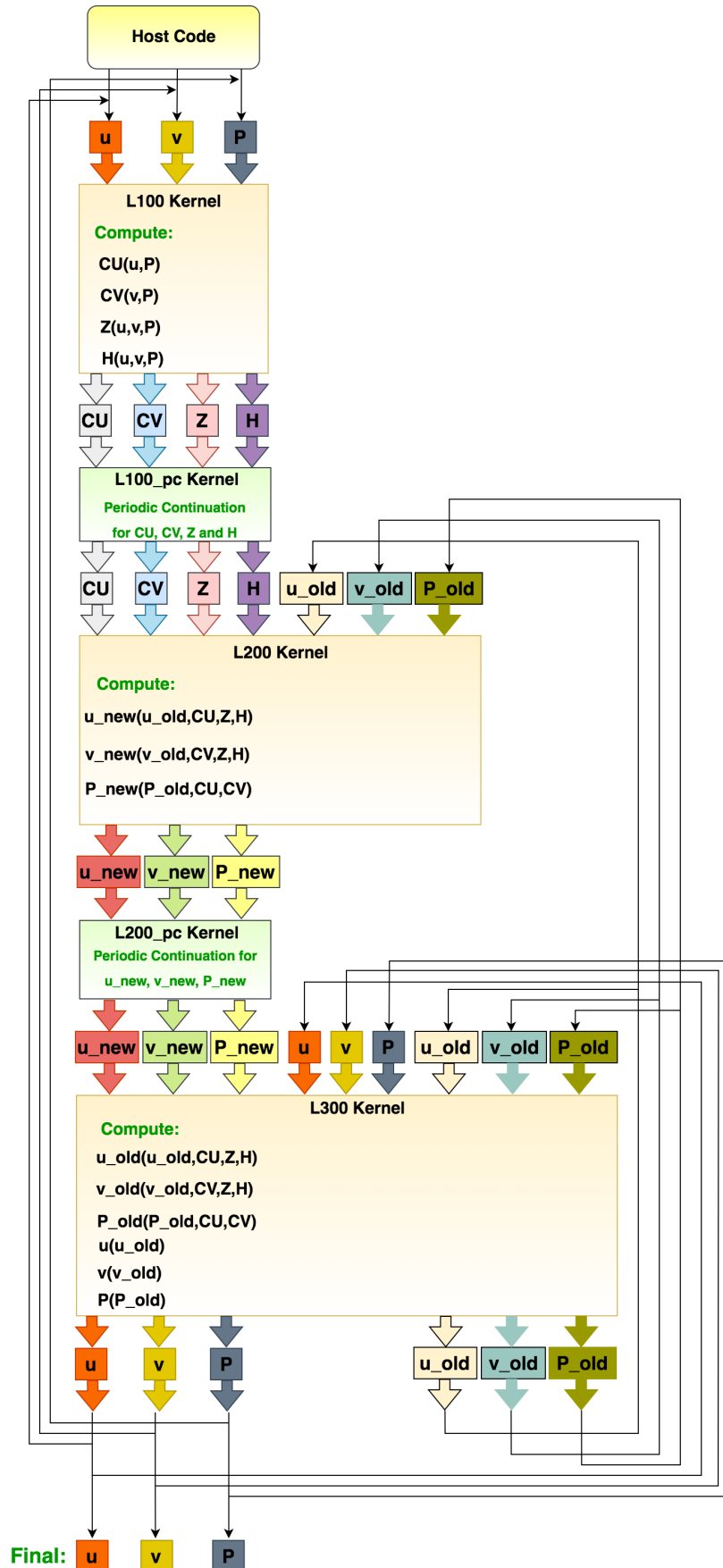


Figure 2.10: A highlight of the data flow of the five main kernels in the SWM coding algorithm.

2.6.2 LFRic Weather and Climate Model mini-app

The **LFRic** is a new atmospheric weather forecasting and climate simulation model that uses a cube-sphere grid to cover the globe, see Figure 2.11. LFRic has been developed by the Met Office in the UK in partnership with universities and other research centres and builds upon the GungHo dynamical core [3] with the aim of providing portable performance achieved using an innovative, architecture-independent, domain-specific programming methodology implemented using PSyclone [AFH⁺19]. The model used in this thesis is an **LFRic** mini-app model consisting of simple dynamics and individual kernels. This version was profiled on a Cray XC40⁶, in the Met Office collaboration system running on a single node. The profiling showed that around 50% of the CPU time was spent on the Helmholtz solver used in the pressure update computation within each integration time-step. The solver performs double-precision matrix-vector multiplication on finite element cells within an outer loop that runs over an atmospheric column of forty vertical levels in the mini-app, see Figure 2.11. The grid in Figure 2.11 is a test grid which is a very coarse representation of the globe. This cube-grid has six faces where each face consists of 12x12 finite-element cells, making 864 cells in the horizontal. Since columns of cells share edges, they cannot all be updated in parallel. A graph colouring scheme is used in LFRic to resolve some of the dependencies updates across the horizontal mesh [AFH⁺19]. Six colouring groups represent the 864 cells. A single "colour" has no dependencies and can be processed simultaneously. The mesh cells are distributed to four groups with 205 cells each plus a 32 cell group and a 12 cell group.

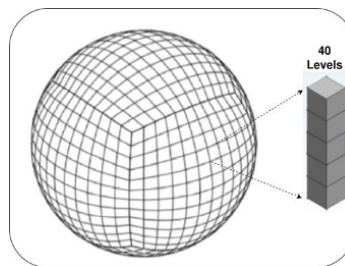


Figure 2.11: “cubed-sphere mesh as used in GungHo with 12x12 subdivisions per face, referred to as a C12 mesh. This gives 864 columns of cells” [AFH⁺19].

⁶The profiling results and the mini-app model was provided by Mike Ashworth at the University of Manchester.

LFRic Matrix-Vector Multiplication Kernel (*MatVec*-kernel)

Listing 2.2 shows the *restricted* C kernel of the matrix-vector multiplication kernel that has been extracted from the LFRic mini-app model, which is used as the basis for the Chapter 7 and 8 in this thesis. The kernel computes a set of 40 (NK) matrix-vector multiplications corresponding to the 40 finite element cells within a single vertical column of a coarse resolution global atmospheric model. Each update consists of a *matrix* of size 8 vertices by 6 faces and a 6 element right-hand-side vector, *x*, producing a left-hand-side output vector of 8 elements, *lhs*. Thus, there are $(8+6+48) * 864 * 40 * 8B = 17$ MB of input data and $8 * 864 * 40 * 8B = 2$ MB of output data for the entire mesh. The size of the matrix is derived from the order of the finite element scheme used.

Listing 2.2: *restricted* C kernel for the Matrix-vector multiplication for NK vertical level

```
#define NDF1 8
#define NDF2 6
#define NK 40
#define MVTYPE double

int matvec_8x6x40_vanilla (
MVTYPE matrix[NK][NDF2][NDF1],
MVTYPE x[NDF2][NK],
MVTYPE lhs [NDF1][NK]
) {
    int df,j,k;
    for ( k=0;k<NK;k++ ) {
        for( df=0; df<NDF1;df++ ) {
            lhs[df][k]=0.0;
            for( j=0;j<NDF2;j++ ) {
                lhs[df][k]= lhs[df][k]+x[j][k]*
                    matrix[k][j][df];
            }
        }
    }
}
```

LFRic Matrix-Vector Multiplication Kernel (*MatVec*-Host)

The **LFRic** mini-app host code prepares the input data for the *MatVec* kernel and reads back the results. In addition, *lhs* appears in the design as input and output, the *MatVec* kernel code only computes the matrix-vector product, and the host code on the ARM CPU updates the *lhs* output data. Depending on the FPGA design, the host code organizes the mesh cells distribution and passes the correct number of cells to the FPGA implementation. Moreover, manages the *MatVec* kernels' data decomposition and execution.

2.6.3 Summary

This section presented a necessary background about the used two weather and climate models in this thesis. Further implementation details are provided in the Chapters 4 and 7.

2.7 Related Work

Raising the interaction level of scientific programmers with FPGAs to a level where they can utilize FPGAs efficiently and rapidly has been challenging. HLS tools have been introduced to help facilitate FPGA's usability and raise the interaction level. The level of abstraction of HLS tools is higher than that of hardware description languages such as Verilog, VHDL and Bluespec. Although the emergence of HLS tools has raised the FPGA's programmability abstraction level, achieving high performance using them is challenging [BRS13] [NSP⁺16] [CFH⁺18] [SVK] [PKB⁺16] [GLN⁺14] [CBM⁺18] and poses usability challenges to scientific programmers.

As stated in Section 2.2, there are now many proposed academic and commercial HLS tools which differ in the level of FPGA control provided and require different levels of FPGA hardware knowledge. In addition, the level of complexity encountered during software development varies between them, as they provide different programming languages and different options and methods controlling FPGA design mechanisms. These differences increase the number of trade-offs between the available options and approaches, affecting the desired performance and increasing the tool's development complexity, and, hence, affect the programmer productivity. For example, accelerating large HPC applications on heterogeneous systems (with single or multiple FPGA) using an HLS tool involves several FPGA-related design challenges and

decisions, such as: multiple kernels implementation, concurrency mapping problem, trade-offs between optimizations options and design decisions. In addition, designs build times can be hours or days and this raises challenges for software programmers as it's a very different way of working. The level of these challenges and the design complexity vary from one tool to another, which adds another layer to the FPGA usability challenge.

Researchers have been exploring and studying the challenges of utilizing HLS tools for accelerating various HPC workloads to find efficient methods to support design decisions and achieve high performance, thereby improving FPGA usability. The effort in studying HLS tools to accelerate HPC workloads on FPGA span multiple areas of the literature. This section provides a literature review of the different approaches used, exploratory studies using HLS tools, optimization method studies, and, finally, frameworks and methodologies proposed to facilitate the use of FPGAs with HLS tools. This thesis applied an exploratory concurrency mapping study in the domain of weather and climate applications, and also a comparative study between higher-level HLS tools (SDSoC OpenCL and SDSoC C++) and a lower-level HLS tool (Vivado HLS) as a contribution to research exploring FPGA usability using HLS tools to the problem of accelerating large HPC applications. This section places this research in the context of the related work.

The related work in the literature can be divided into three categories: Single HPC-Kernel accelerators, Exploratory studies with HLS tools, HLS tool Comparison and Survey studies.

2.7.1 Single HPC-Kernel accelerators

To contribute in addressing the challenges raised from the emergence of HLS tools, authors have proposed performance-tuned accelerators using different HLS tools such as Alter OpenCL SDK, Maxeler, OmpSS, Xilinx Vivado and Xilinx SDAccel and SDSoC for different HPC workloads such as in weather and climate, linear algebra, machine learning and finance.

For example, the authors in [ZPM18, ARAM19, SKN⁺16] presented performance-tuned FPGA accelerators for HPC workloads from weather and climate applications. [ZPM18] proposed a high-performance 2D/3D stencil computation accelerator using the Intel OpenCL SDK and implemented it on an Arria 10 GX 1150 FPGA. The proposed FPGA design combined spatial and temporal blocking to avoid issues caused

by the input size restrictions and by employing a set of FPGA-specific optimization techniques, such as loop collapsing, exit condition optimization, and padding. In [ARAM19] the authors used the Xilinx Vivado environment to propose an FPGA design for an LFRic workload kernel. Their design consists of twelve IP blocks (Spatial Parallelism) implemented on a Xilinx Zynq UltraScale+ board. The authors in [SKN⁺16] utilised the Altera Stratix V 5SGXEA7 FPGA and system-on-programmable-chip development tool to propose an efficient FPGA hardware design for the 1D tsunami simulation. The design depends on streaming computation where fusion loop, shift buffers and cascading processing elements optimisation techniques are used to improve the performance.

Across almost all scientific areas in HPC applications, linear algebra operations such as partial differential equations are ubiquitous. Studies such as [DVKG05, KJPN10, FOS⁺14, BFV⁺17, MML] presented FPGA performance-tuned accelerators for linear algebra workloads. Linear algebra implementation on FPGAs has focused mostly on matrix-matrix multiplication, sparse Matrix-vector multiplication, and simple iterative stencil-based solvers. For matrix-multiplication, the authors in [DVKG05] have proposed a block design accelerator that enhances data locality and re-usability considering the local storage and I/O limitations in FPGAs. Another example is [KJPN10] which provides two FPGA accelerator designs that support IEEE 754 double-precision floating-point matrix multiplication on Virtex-5 FPGA. For sparse matrix-vector solution, authors in [FOS⁺14] proposed a novel optimized sparse matrix-vector FPGA design that exposes parallelism across rows with low usage of on-chip memory. Another example is provided by authors in [BFV⁺17]. They have presented an FPGA design for the matrix multiply benchmark. The design is implemented on a Xilinx Zynq Ultrascale+ board and synthesised using the OmpSS approach. Authors in [MML] demonstrated an accelerator design for a large matrix multiplication kernel using the Maxeler platform. They have utilised the data flow engine MAX3 card based on a Virtex 6 FPGA chip.

Other examples of performance-tuned HPC accelerators in the literature are FPGA designs provided using the Xilinx SDSoC and SDAccel. For example, authors in [HM21] utilised the Xilinx SDSoC C/C++ tool to create a deep convolutional neural network accelerator implemented on the Xilinx ZYNQ Ultrascale ZCU104 board. The same authors provide another design in [CCS⁺18] for image processing. They have used the SDSoC C/C++ development environment and Xilinx Zynq-700 to create an FPGA design for the Gaussian blur function. Authors in [MWT⁺20] proposed an optimized

FPGA design for image processing using SDSoc C/C++ development software and the OpenCV image library (XfOpenCV). The design is evaluated using several image processing algorithms, and the results are presented based on the processing speed, development cycle and power efficiency. The authors in [KG17] proposed an FPGA accelerator for the AKAZE feature detection algorithm. The design is synthesized using the Xilinx SDAccel and implemented on a Xilinx Virtex-7 FPGA. The study also discussed different optimization strategies and methods for reaching the best design.

These studies targeted FPGA accelerator design for a single HPC kernel in isolation. In contrast, this thesis ultimately targets the context of multiple kernels of HPC applications that will require large distributed accelerators running on a heterogeneous machine.

2.7.2 Exploratory studies

Exploratory studies are another approach in the literature to address the challenges of using HLS tools with FPGAs. Authors have explored the different options of optimization methods and design decisions available with the HLS tool's programming language or the vendor-based FPGA mechanisms. These studies resulted in optimizations, recommendations, proposed methodologies and frameworks based on the selected HLS tool and the target HPC workload.

For example, authors in [JZ16,CFH⁺18,PFC20,VHKF16,GF20,LWY⁺17,SEEZ19] provided studies exploring the OpenCL programming language optimizations in accelerating HPC workloads. Different HLS tools are explored, and different HPC kernels are utilized. These studies explored a similar set of OpenCL-based optimizations, such as OpenCL local memory and vectorization. However, the aim of the exploration, tools and HPC kernels are different.

Authors in [JZ16] explored OpenCL code optimizations for **stencil kernels**, to improve stencil computation kernels in both OpenCL single task and NDRange modes. They have utilized an Altera FPGA and discussed the performance details of the Altera FPGA memory system. They recommended utilising the constant memory and shift register patterns with the OpenCL single task kernel. Moreover, applying constant memory, local memory and vectorisation to the NDRange kernel can provide higher memory access bandwidth, hence better performance. Similarly, in [CFH⁺18] study, authors explored a few simple steps to improve the usability of the HLS tool and help achieve high performance for **stencil computational kernels** on FPGA. They proposed a “best-effort” guideline consisting of five main HLS optimization strategies:

explicit data caching, customized pipelining, double buffering and scratchpad reorganization for developing FPGA accelerators in an HLS environment. The authors have quantitatively evaluated the best-effort optimizations methodology and illustrated its match to software programming techniques, using accelerators from the MachSuite benchmark, and a Xilinx Virtex-7 FPGA with the Xilinx SDAccel HLS tool. The proposed guideline is only tested for MachSuite benchmark accelerators; therefore, exploration for a wide range of HPC workload and HLS tools is still required.

In contrast, the study from [PFC20] explored the variations in **OpenCL coding styles** and resulting changes in performance of OpenCL kernels implemented on Xilinx Kintex UltraScale XCKU060-2 FPGA. The explored OpenCL coding styles included: task and NDRange kernels, vectorisation, on-chip local memory and burst mode. A k-means algorithm was used as a case study, where they have produced ten k-means OpenCL code versions and twelve integer data sets. They evaluated the effects of different data set characteristics, the number of processing cores, resource usage, and performance.

Another exploratory study example is provided in [VHKF16]. In this study, authors employed the OpenCL kernels through taking the considerations of **the hardware constraints** which can improve the FPGA design performance. The authors have evaluated general optimization techniques in OpenCL, such as single task and NDRange kernels, single-item and work-group, static coalescing, manual vectorization and intra-kernel channels on the OpenDwarfs benchmark Suite, using Altera FPGA. Their evaluation focused on the effectiveness of the identified optimization techniques in terms of performance and resource utilization. They revealed that the FPGA accelerator developed using the OpenCL kernels showed improved performance, thereby showing the potential of OpenCL for designing efficient FPGA accelerators. In contrast, authors in [SEEZ19] have evaluated the performance potential of a specific OpenCL to FPGA optimisation methods that they claim are **not being deeply assessed** in the literature. Those optimisations are: Local atomic operations, OpenCL single-task kernel, avoiding global atomic operations, using channels/pipes between producer and consumer kernels, multi-threaded consumer and single-task consumers, and appropriately specifying work-group sizes. The evaluation was conducted using micro-benchmark workloads, and the FPGA designs were implemented on Altera Stratix V GX FPGA and compared to implementations on an Intel 2160 CPU.

The authors in [GF20] and [LWY⁺17] explored various OpenCL optimisation techniques to address the **memory-access bottleneck**. In [GF20] the memory-access bottleneck is explored on the dense graphs breath-first search (BFS) building block. They have explored three categories of optimisations: OpenCL-specific optimisations, architecture aware optimisations and application-specific optimisations. The study has considered the choice of data structure, such as queue versus array, the number of memory banks and the kernel launch configuration. The OpenCL-specific optimisations used are task kernel, kernel fusion, elimination of host-based synchronisation and maintaining the data in local memory to avoid expensive global-memory accesses. The architecture-aware optimisations were: local queue, multiple memory banks and speculated iterations. The application-specific optimisations merged the three computational stages of BFS (filter-apply-expand) into a single kernel to avoid overhead from kernel launch and eliminate duplicate entries. In citeluo2017evaluating the authors studied the performance/energy effects of the irregular memory access patterns to off-chip memory in OpenCL to FPGA. A case study kernel called XSbench from the Monte Carlo model was used to evaluate the experiments in the study. They transformed and optimized the XSbench kernel in OpenCL using a range of optimizations and implemented it on an Altera Arria10 FPGA using the Intel OpenCL SDK tool.

Another exploratory study example was carried out on the Vivado HLS tool in [BD19]. The authors in this study explored a design improvement for an existing Vivado design for an advection scheme to reduce the data movement overhead. The authors explored the dataflow style to redesign the existing Vivado design. They have first developed a profiling approach that can be used within the Vivado HLS tool to highlight the source of bottlenecks in a Vivado FPGA design. They have found that 86% of the existing kernel design's execution time was spent on DRAM data access. Therefore, to improve the kernel design, they have improved two main areas: the kernel runtime and the DMA access time. The kernel code is redesigned into four functions (read, prepare-stencil, compute, write) and the Dataflow HLS pragma is used to launch the four functions in parallel. This design decision led to data streaming between the execution stages and improved kernel latency. For improving the DMA access time, they have defined a C Struct function to increase the data access width to four double-precision values per access, and divided the data into chunks, overlapping them with the compute stage where possible. A Xilinx Kintex Ultrascale KU115-2 FPGA card and 18 cores Broadwell CPU was used to evaluate the proposed design. This work suggested that data

movement cost is a key performance factor for achieving high performance FPGA designs. In addition, their work involved low-level hardware design details that are not feasible to be accessed in higher-level HLS tools such as Altera, SDAccel and SDSoc.

The studies mentioned above focused on exploring the OpenCL language and the FPGA-vendor HLS tools, but for different aspects, related to the target HPC kernel and the exploration study's aim. These studies have not discussed other OpenCL design options, such as Dataflow and multiple kernels mapping. In addition, the explorations have not discussed the concurrency mapping problem in multiple kernels, which is a focus of this thesis focus.

2.7.3 Comparison and Survey studies

HPC Scientific programmers face another phase of complexity in utilising HLS tools which is related to the various options available in HLS tools. Some of these tools are only compatible with a specific set of FPGA boards or work within a particular environment. This leaves the programmer with only a small choice among the available HLS tools. However, deciding the best choice is always challenging. These tools vary in their abstraction level presented to the programmer and require different levels of hardware knowledge; hence, different levels of outcome in productivity result. To improve the HLS choice decision, researchers have conducted comparison and survey studies.

For example, authors in [MVBG⁺12], and [SW19] provided a qualitative comparison and survey study for several selected HLS tools. In [MVBG⁺12] the study aims to provide designers with a view on the tools and help the designer to decide the best choice that suits their needs. The comparison was carried out based on three main metrics: the capabilities of the tool, quality of results and usability. The list of tools was Xilinx AccelDSP, Agility compiler, AutoPilot (Vivado), BlueSpec, Catapult C, Compaan, C-to-Silicon, CyberWorkBench, DK Design Suite, Impulse CoDeveloper, ROCCC, Synphony C Compiler. In [SW19] the authors demonstrated a Design Space Exploration (DSE) study to overview and classify published HLS tools. This study compared the different approaches to provide details of their limitations, trade-offs and produce a guide for researchers wanting to create their own HLS DSE. Moreover, the main techniques proposed in each tool are summarised.

With a similar aim, the authors in [NSP⁺16] and [PZMM17] provided a comprehensive analysis study for HLS tools, however only for a small set of recent HLS tools. In [NSP⁺16] the authors proposed a methodology for evaluating HLS tools, and they

used them to evaluate the selected HLS tools. The evaluation study is conducted on one commercial and three academic HLS tools (DWARV, BAMBU, LEGUP), and the experiments are carried out on a common set of C benchmarks and implemented on Xilinx Stratix V and Virtex-7 boards. The tools are compared in terms of the performance metric (maximum FPGA frequency, cycle latency, wall-clock time) and the use of resources. Similarly, the authors in [PZMM17] provided a comparison study and overview for three different FPGA programming methods. The three HLS tools used are LegUp, Quartus and Intel OpenCL SDK. Six kernels from the Rodinia benchmark suite were used and implemented on a Stratix V FPGA to provide a quantitative performance evaluation of the three HLS tools. The authors have identified bottlenecks in the LegUp HLS tool and recommended a set of improvement methods.

In contrast, the authors in [LY16] provided a study for comparing low-level RTL programming vs HLS tools. They reported the development of kernels in the Vivado HLS environment to explore the programmability transition from an FPGA low-level programming language (Verilog) to a higher level of abstraction methodology (Vivado HLS). Their exploration involved the implementation of a filter example to show the step-by-step creation of an FPGA design in the Vivado HLS environment. Verilog design and Vivado HLS were compared in terms of their development time and it was found that the implementation in the Vivado HLS approach required one week compared to two weeks taken by the RTL approach. This study showed the importance of the HLS tools in reducing the effort in FPGA programming to help software programmers who come new to FPGAs. However, the study did not discuss the tool performance benefits and challenges against other HLS choices. The authors in [Ken19] also provided a comparison study for two HLS tools. However, their study focused on two HLS tools using the same programming language: OpenCL-based Xilinx SDAccel, and Intel FPGA SDK. This study aimed to test the portability in both OpenCL-based FPGA designs since they build upon the same programming model. The study shows that OpenCL-based FPGA design portability is possible by following design patterns that work well for both tools.

A comparison study is also undertaken in this thesis. However our study compares the implementation of a relatively large HPC application using a low-level of abstraction HLS tool (Vivado HLS) versus using higher-level of abstraction HLS tools (SDSoC OpenCL/C++).

2.7.4 Related Work Summary

This thesis contribution fits in three areas of the overall body of related work: This thesis ultimately targets the context of multiple kernels of HPC applications that will require large distributed accelerators running on a heterogeneous machine. In addition, the study focused on exploring the design options and their effects on the performance, the generated hardware, and the programmability, rather than producing fine-tuned accelerator. This thesis also explores and compares the implementation of two relatively large HPC applications using different HLS tool methodologies (Vivado HLS versus SDSoC OpenCL and SDSoC C++).

Chapter 3

Research Methodology and Study Experiments Setup

This chapter presents the methodological approach that is used in this thesis. The used methodology consists of two primary research studies, which are referred to as an *Exploratory Study* and a *Comparison Study*. Those two studies are carried out for two HPC-based benchmark applications: SWM and the Matvec kernel from the LFRic-mini-app. The first part of this chapter details the target FPGA board used to achieve the research objectives. The second part presents details of the *exploratory* study method. The third part discusses the *comparison* study method and components.

3.1 Target FPGA Hardware

The targeted FPGA platform is the Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [Xil20]. We have selected this FPGA hardware since it is a SOC where the ARM CPU and FPGA fabric are built within the same hardware chip. This thesis study focused on implementing the exploration experiments on SOC FPGA, where the requirement for PCIe data movement is neglected. The ZCU102 board consist of two main parts: a Processing System (PS) and a Programmable Logic (PL). The PS part has the fundamental components to run general-purpose tasks, particularly the Host-code of an HPC application. For example, the application Processing Unit (APU) has quad-core ARM Cortex-A53 with 32KB L1 Cache and 1MB L2 Cache. The ARM CPUs execute the Host-code and control the accelerator (i.e. the HPC FPGA kernel) design in the PL part. Other peripherals include a Real-Time Processing Unit (RPU),

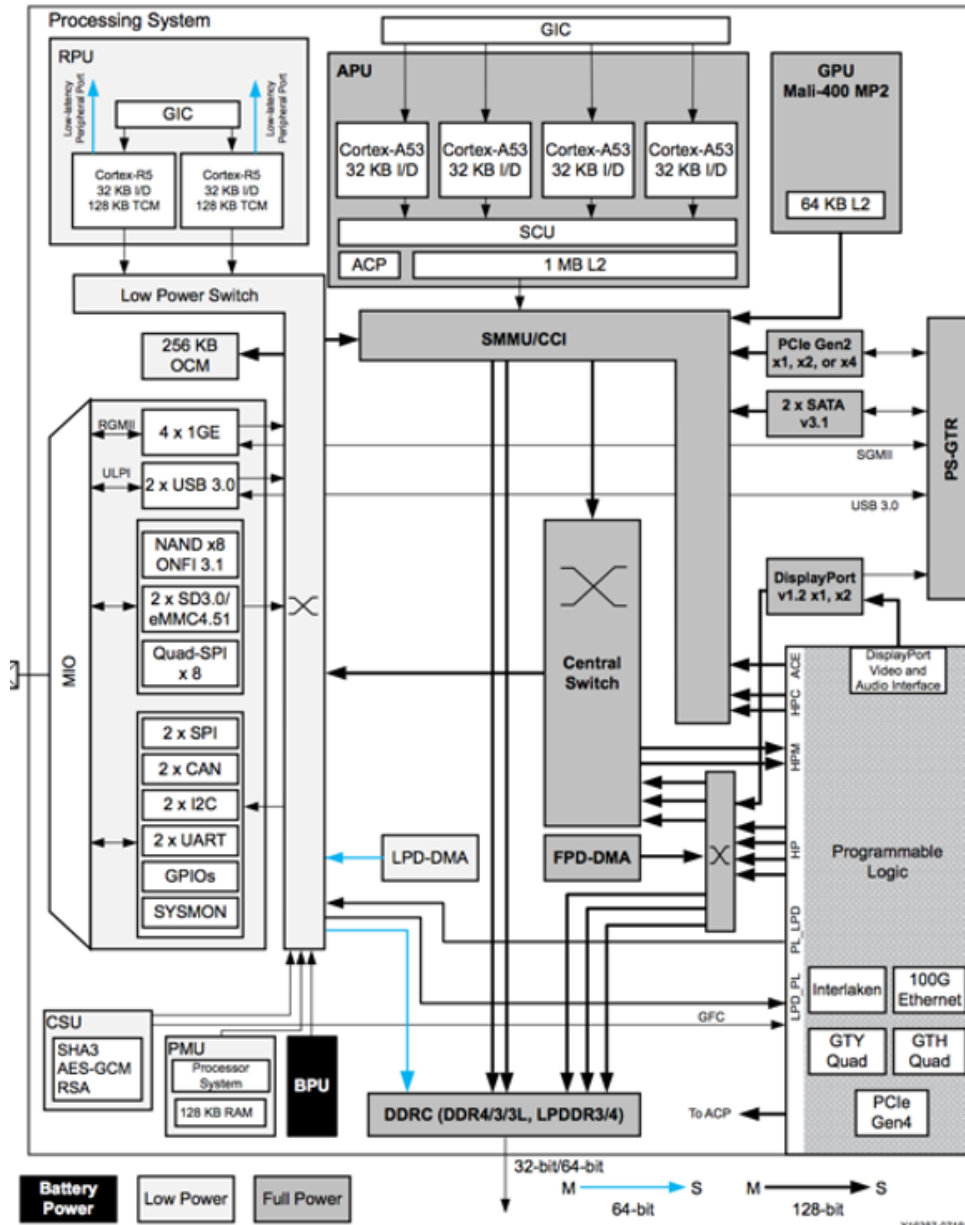


Figure 3.1: Zynq UltraScale+ MPSoC Top-Level Block Diagram [Xil20].

an ARM Mali-400 MP2 GPU, High-Speed Connectivity and Dynamic Memory Controller (DDRC) for communication management between the PS and PL. The board also has 4GB DDR memory, shared between the CPU and the FPGA, consisting of four memory banks supported with four access ports, as shown in Figure 3.1. The PL part contains the Zynq UltraScale XCZU9EG-FFVB1156 FPGA and has all the logics used to build FPGA accelerators. Table 3.1 shows that the XCZU9EG-FFVB1156 FPGA resources consist of 548,160 *Flip-Flops* (FFs), 274,080 *Look-Up-Tables* (LUTs), 912 *Block RAMs* (BRAMs) and 2,520 *DSP Slices* (DSPs).

The communication between the PS and the PL parts is managed through 12 Master and Slave interfaces, see Figure 3.1. The Master interfaces are M_AXI_HPM0_LPD and M_AXI_HPM[0-1]_FPD. The ARM CPU uses the Master interfaces to control the accelerator by writing/reading to control registers. Each accelerator kernel has some control registers in a memory accessible by the ARM to control the Kernel's start/finish and data movement. The PL part uses the Slave interfaces to retrieve/send data to the ARM CPU by writing to the shared DDR memory. The slave interfaces are S_AXI_HPC[0-1]_FPD, S_AXI_HP[0-3]_FPD and S_AXI_LPD. AXI protocols [Xil21a] manage the communication through those interfaces. There are three AXI protocols:

- AXI4 for high-performance memory transactions between the Kernel and the Shared DDR memory.
- AXI4-Lite, a protocol for kernel control and status.
- AXI4-Stream for high-performance data streaming.

The choice of those interfaces and protocols is the responsibility of the compiler in SDSoc designs, but has to be managed manually by the developer in the Vivado tool, as we will discuss in chapters 6 and 7.

Table 3.1: Xilinx Zynq UltraScale+ MPSoC ZCU102 board Available Resources Count.

Hardware Resources	Resource Count
CLB Flip-Flops	548160
CLB LUTs	274080
Block RAM Blocks	912
DSP Slices	2520

3.2 System and Experimental Setup

The version of the utilized HLS tools (SDSoC OpenCL, SDSoC C++ and Vivado) is 2018.2. The ZCU102 board ARM CPU runs Ubuntu 16.04.5 with the device tree of the SDSoC OpenCL and Vivado environment, the binary files of the FPGA boards and the necessary drivers and libraries. For the SDSoC C++ designs, we used the Bare-metal [Xil21b] method to execute the designs on the FPGA board. All reported results are averaged over five runs.

3.3 Exploratory Study

The *exploratory study* is an exploration of the concurrency mapping techniques, low level optimisations and the different mapping option trade-offs that are available from the HLS tool level and from the programming language. In this thesis we conducted two exploratory study where one targeting the SWM application and the other targeting the Matvec application. These two exploratory studies are designed to collect data on performance and resource usage, along with information related to programmability, from each of the two studies, independently. Comparison between and across the results gathered is the subject of the Comparison Study described in Section 3.4.

3.3.1 Exploratory Study (1): SWM Concurrency Mapping Exploration

In the first exploratory study, we explore FPGA mapping techniques for mapping the concurrency levels available in the SWM benchmark to the ZCU102 FPGA board, using the SDSoC OpenCL and the Vivado HLS tools. There are two levels of concurrency in the SWM application; Concurrency within each of the SWM kernels (Single kernel) and concurrency between the SWM kernels (Multiple kernels). We explored these two concurrency levels and split them into two parts. The following points present these two study parts:

- **Exploratory Study (1) Part One: L100 kernel Concurrency Mapping:** The first part of the exploratory study (1) explores the concurrency within a single SWM kernel mapping. We chose a candidate kernel out of the nine SWM kernels to study its concurrency types mapping to approach this study. The L100 kernel, described in Section 2.6.1 from Chapter 2, has been chosen to conduct

the concurrency mapping exploration to a single FPGA board because it is a good candidate kernel that represents typical levels of concurrency in an HPC kernel. The study starts with exploring the options available in SDSoC OpenCL to map the L100 kernel's concurrency. The mapping options are identified from the language level (OpenCL) and at the FPGA level (SDSoC HLS tool). These options are then explored and characterised based on their resulting performance and FPGA resource usage. Secondly, the L100 concurrency levels are explored in the Vivado HLS tool after identifying the mapping options available in the Vivado tool, and the study then gathers these mapping options' resulting performance and resources usage. In addition, the study addresses information related to the development effort required in both HLS approaches to provide insight to the traditional HPC software developer, guiding their future design choices and trade-off options between the mapping options. This study part (one) is presented in **Chapter 4**.

- **Exploratory Study (1) Part Two: SWM Multiple-Kernels Mapping:** The second part of the exploratory study (1) explores the mapping of the concurrency between the SWM kernels. This exploration includes studying the available options and trade-off choices in the number of kernels that can fit in a single FPGA, the options for data exchange management between the kernels, the problem size choice, and the impact of optimisation levels that can be applied. In addition, the study explores the application of the optimisation lessons learned from mapping the L100 kernel concurrency to the other SWM kernels. The exploration for mapping the multiple kernels SWM application to a single FPGA is conducted using the SDSoC OpenCL, and Vivado approaches. This study part (two) is presented in **Chapter 5**.

3.3.2 Exploration Study (2): *MatVec* Kernel with SDSoC OpenCL/C++ and Vivado HLS

In the second *exploratory study*, we explore the implementation of an existing FPGA design of a key kernel (referred to as *MatVec*) taken from the LFRic Weather and Climate model developed using a low-level HLS tool, Vivado, and gathers data for several designs created using the relatively higher-level approaches of SDSoC OpenCL and SDSoC C++. The study includes exploring the techniques available at a higher level of abstraction (in SDSoC OpenCL and SDSoC C++) with the aim of replicating

the design decisions made in the low-level Vivado HLS tool as closely as possible. In addition, this study explores and gathers qualitative data for implementations designs, again with the aim of matching as closely as possible, the *MatVec* design in the Vivado HLS. This study is presented in **Chapter 7**.

3.4 Comparison Study

The *comparison study* is a comparison of the gathered data from the two *exploratory studies*. This study compares performance, resource usage, and programmability differences between the best implementations of the two weather and climate benchmarks, SWM and MatVec, using the different HLS approaches. Two comparison studies are conducted which the following Subsections present.

3.4.1 Comparison Study (1): SWM Implementations In SDSoC OpenCL versus Vivado

The first comparison study compares the gathered data from the mapping exploration study (1) of the L100 kernel and the SWM multi-kernels between the SDSoC OpenCL and the Vivado. This comparison compares the best implementations results, methods, mapping techniques and trade-offs of the concurrency mapping explorations from the two different levels of abstraction of the two HLS tools. This study is presented in **Chapter 6**.

3.4.2 Comparison Study (2): *MatVec* Kernel implementations in SDSoC OpenCL and C++ Versus Vivado HLS

The second comparison study compares the best implementations of the MatVec kernel from the three HLS approaches (SDSoC OpenCL, SDSoC C++ and Vivado) that we explored in the exploratory study (2). This comparison compares the gathered data of performance results, resources usage, data movement methods and differences in design decisions and applied design techniques. This study is presented in **Chapter 8**.

3.4.3 Comparison Study Metrics

In the two comparison studies we evaluate the gathered data from the two exploratory studies through the use of quantitative and qualitative metrics. These comparison metrics are chosen based on the research objectives aims, which interested in implementations performance, resource usage, data movement methods and HLS approaches programmability. The following points present the comparison metrics:

- **Performance:** This metric compares the performance in terms of the kernel's computational time and overall execution time. The kernel's computational time is measured in two forms: the arithmetic computation *flops per second rate* that the design produces and the runtime (seconds). Flops rate in a design is calculated by knowing the number of floating operations that the design needs. The runtime (seconds) is used to report the multiple SWM kernels implementations performance (Chapter 5), while the flops form used for the L100 kernel and the LFrisc benchmark best implementations. The kernel *overall execution time* is the data movement time added to the kernel's computational time (in seconds). The data movement time is calculated based on the bytes per second.
- **Resource usage:** This metric is used to compare the percentage of FPGA resources that a design has consumed.
- **System Hardware design:** This metric compares the generated hardware system designs differences. We compare the differences in terms of the chosen IP blocks, communication ports, data movement methods, interconnection techniques. In addition, we compare the coding choices that led to the generated system design.
- **Data movement method:** This metric compares the data movement time in bytes per second between the applied data movement methods. In addition, we discuss and compare the available options for exploiting the memory hierarchy and how the choice affects performance. In the Matvec design, we also compare how close to the Vivado design data movement design the SDSoC OpenCL and SDSoC C++ data movement methods get and why.
- **Development effort:** This metric is a qualitative review of the steps and effort of developing the design in each development environment: SDSoC OpenCL, SDSoC C++ and Vivado.

- **Level of hardware expertise required:** How much hardware knowledge the traditional HPC programmer needs to achieve "good" performance compared to the three development environments: SDSoc OpenCL, SDSoc C++ and Vivado.

3.5 Summary

This chapter presented the methodological approach to achieve the thesis objectives. The chapter presented first the target ZCU102 FPGA board and the system setup. Following that, the two primary research studies (Exploratory and Comparison studies) were explained, and how they will be approached in the next chapters are also discussed.

Chapter 4

Exploratory Study (1) Part One: L100 kernel Concurrency Mapping

This chapter presents the first part of the *exploratory* study (1) for mapping the concurrency available in a Single kernel (L100) from the SWM application to the Xilinx ZCU102 FPGA board using the SDSoC OpenCL and also the Vivado HLS tools. It discusses the trade-offs involved and the performance results and describes the different implementations for mapping the single SWM candidate kernel. This chapter starts by presenting the concurrency levels related to the mechanisms available in the HLS tools and programming languages and the concurrency types available in the L100 kernel. In addition, the different possible coding options for writing the L100 kernel are also presented. We then present the study setup that involved the basic optimisations applied in each mapping mechanism experiment. Section 4.3 then presents the exploration study for mapping the L100 kernel using the SDSoC OpenCL; While Section 4.4 presents the mapping of the L100 using the Vivado approach. Finally, the chapter ends with a section summarising the main lessons learned from the experiments carried out.

4.1 L100 Concurrency and Coding Options

Mapping a kernel's concurrency levels efficiently to the FPGA using HLS tools involves multiple design decisions and option choices related to three areas. *First* is the **programming language constructs** e.g. OpenCL. *Second* is the **vendor computational and data-movement optimisation mechanisms**, and *Third* is the different ways for **writing the kernel code(s)**. The software developer has to have a rationale

for choosing among all the different available options. This section explores the concurrency mapping options available in the *l100* kernel using first SDSoC OpenCL and secondly Vivado HLS. In addition, a characterisation of their use in terms of performance and FPGA resource usage, and address questions related to development effort. The SWM kernels, such as *l100*, *l200* and *l300*, have similar concurrency levels. The focus in this section is on studying the *l100* kernel because it represents most of the examples of concurrency levels in the SWM application. The lessons learned from this exploration can then be applied to the other kernels in the SWM multiple-kernels mapping exploration study in Chapter 5.

As described in Section 2.6.1 from Chapter 2, the L100 kernel takes three arrays as input (u, v, P) representing the wind velocities and the potential pressure. These are used to calculate four arrays (CU, CV, Z, H) representing the mass fluxes, the potential vorticity and the surface height. There are several ways to express this kernel algorithm in a high-level programming language such as C++ and this initial expression of the computations in code constrains the choices to exploit concurrency. Here, we consider three candidate coding options as represented in Figure 4.1. The calculations can be

<pre>//Compute cu,cv,z and h for i in 0, M; j in 0, N : cu[i+1][j]=.5*(p[i+1][j]+p[i][j])* u[i+1][j]; cv[i][j+ 1]=.5*(p[i][j+1]+p[i][j])* v[i][j+1]; z[i+1][j+1]=(fsdx*(v[i+1][j+1]-v[i][j+1])- fsdy*(u[i+1][j+1]-u[i+1][j]))/(p[i][j]+ p[i+1][j]+p[i+1][j+1]+p[i][j+1]); h[i][j]=p[i][j]+.25*(u[i+1][j]*u[i+1][j]+ u[i][j]*u[i][j]+v[i][j+1]*v[i][j+1]+ v[i][j]*v[i][j]);</pre>	<pre>//Compute cu,cv,z and h for i in 0, M; j in 0, N : cu[i+1][j]=.5*(p[i+1][j]+p[i][j])* u[i+1][j]; for i in 0, M; j in 0, N : cv[i][j+ 1]=.5*(p[i][j+1]+p[i][j])* v[i][j+1]; for i in 0, M; j in 0, N : z[i+1][j+1]=(fsdx*(v[i+1][j+1]-v[i][j+1])- fsdy*(u[i+1][j+1]-u[i+1][j]))/(p[i][j]+ p[i+1][j]+p[i+1][j+1]+p[i][j+1]); for i in 0, M; j in 0, N : h[i][j]=p[i][j]+.25*(u[i+1][j]*u[i+1][j]+ u[i][j]*u[i][j]+v[i][j+1]*v[i][j+1]+ v[i][j]*v[i][j]);</pre>	<pre>//Compute functions //cu,cv,z and h Compute: cu (u,p); cv (v,p); z (u,v,p); h (u,v,p);</pre>
A- One Nested Loop	B- Four Nested Loops	C- Four Functions

Figure 4.1: Pseudocode for the *L100* kernel coding options. A- Wrap the kernel operations with one *for* loop. B- Wrap each operation in a loop. C- Wrap each operation in a *function*.

wrapped with one nested *for* loop as in the pseudocode in Figure 4.1(a). Another coding option is to wrap the assignment to each variable (CU, CV, Z, H) with a separate nested *for* loop Figure 4.1 (b). A third coding option considered is to implement each of the assignments as a separate function Figure 4.1 (c). As is apparent in Figure 4.1, each assignment, and every computation of an element of each of CU, CV, Z and H , are all independent. This means that, theoretically, with an ideal machine design all

the operations could be executed in one step¹; there is an high degree of concurrency in this *embarrassingly parallel* algorithm.

Three distinct types of concurrency may be identified in L100 kernel: *Instruction-Level Parallelism* (ILP) which involves exploiting parallelism from the computation of each instance of the loop iterations (i.e. each *statement*) since each consists of several floating point operations. This parallelism is clear in the code options (a) and (b) in Figure 4.1. *Functional Parallelism* (FP) where each function in code option (c) in Figure 4.1 can be processed in parallel. *Data Parallelism* (DP) where the processing of the iterations in each loop in the algorithm can be carried out in parallel. In OpenCL terminology, each such iteration (or group of operations) may be considered as a work-item (WI).

FPGAs are a platform where a hardware solution can be tailored to fit the algorithm requirements. The question for the HPC scientific programmer/developer is: how best to map the different concurrency types available in the L100 algorithm to exploit the FPGA's potential, using the mechanisms available in the OpenCL language and in the Xilinx SDSoC (see subsection 4.3) and Vivado HLS tools (see subsection 4.4)?

4.2 Study Setup

The HLS vendor-supplied computational optimisations for either SDSoC OpenCL or Vivado HLS are used for mapping the kernel's concurrency and part of the process of optimising the computation. Background on the available optimisations related to mapping is provided in subsection 2.5.1 in the background chapter. However, applying only the mapping mechanisms to the kernel is not sufficient (i.e. does not generally result in efficient performance) without utilising the optimisation strategies that are presented in subsection 2.5.1 for improving the data-movement required by a kernel. The memory hierarchy that is available in the ZUC102 FPGA must be exploited and memory bandwidth can be increased, for example, by utilising the full range of memory ports for accessing data either in the DDR memory or in the BRAMs.

Therefore, a set of optimisation strategies is proposed which are applied in each mapping mechanism experiment that is carried using SDSoC OpenCL or Vivado HLS, in the following order. This optimisation order was chosen based on the programming complexity of the optimisation. The first optimisation being the more straightforward

¹ The computation of *CU*, *CV*, *Z* and *H* points are not equivalent as some are more complex than others but they can all start in parallel.

optimisation strategy and requires the least programming effort.

1. Enable the max memory ports to increase the bandwidth, leading to a lower overall time to access the data.
2. Transfer data from the DDR memory to the FPGA BRAMs to take advantage of the lower access latency of BRAMs.
3. Transfer the data to the BRAMs in burst-mode, so that we hide the data transfer latency. Bursting mode is where the read (or write) loop is pipelined.
4. Use the array partitioning attribute/Pragma with the pipeline (work-item pipeline attribute with OpenCL NDRange kernels), unrolling and dataflow attributes/Pragmas, to provide more BRAM ports for the calculations, and thus increase the memory bandwidth available.

For both mapping explorations for the `l100` kernel, using SDSoC OpenCL and Vivado HLS, a domain of size 64x64 is used. Whilst a domain size of 64x64 is small, the computation could be undertaken in chunks (similarly to the LFrnc benchmark implementations in Chapter7). A single iteration of the inner loop of the `L100` kernel has 24 32-bit `float` operations. In total, therefore, there are 98,304 `float` ops for the 64x64 domain for a single cycle. The kernel requires 23 `read` accesses, and 4 `write` accesses to the global DDR memory in each iteration (in the *base* version of the code), leading to a total of 368 KiB input data and 64 KiB output data for a single time-step. For all experiments, the highest hardware optimisation flag, `-O3`, is used and the highest clock frequency that is possible without breaching the timing constraints is used. The performance results in this exploration chapter are the kernel compute time only and not overall runtime (which will be reported in Chapter6.)

4.3 L100 kernel mapping using SDSoC OpenCL

Using SDSoC OpenCL, there are two paths available for mapping the concurrency in the *l100* kernel to the ZCU102 FPGA. The first is through the use of the high-level **OpenCL programming constructs** and the second through the use of **the SDSoC mapping mechanisms**. As presented in Section 2.5, there are three critical mechanisms available in the OpenCL language to control/change the mapping of the concurrency in an application to the FPGA. The first is the *OpenCL kernel type* which can be either a *task* or an *NDRange* kernel. The second is *the number of kernels* included

Table 4.1: The performance effects of each optimisation on the L100 kernel in SDSoC OpenCL. (Seq) no optimisations; (Max) max DDR memory ports; (BRAMs) use of BRAM memories; (B) use burst mode; (AP) use array partitioning

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Loop trip Count	Execution Time Seconds	Speedup
L100-Seq	4,714,496	1151	4096	188.4	-
L100-Max	35,457	853	4096	3.4	55.41x
L100-Max-BRAM	97,344	1528	4096	2.29	82.27x
L100-Max-BRAM-B	14,805	524	4096	0.66	285.48x
L100-Max-BRAM-B-AP	14,804	524	4096	0.66	285.48x

in the software design, reflecting the developer's choice as to how to implement the algorithmic operations of the application in one or more kernels. The third is the choice of the type of OpenCL API command queue(s) selected for controlling the kernel's execution, which can be either *in-order* or *out-of-order* queue(s).

4.3.1 SDSoC OpenCL L100 Initial Implementation

The first exploration carried out, in mapping the L100 concurrency using the SDSoC OpenCL approach, explores the impact of applying the data-movement low-level FPGA optimisations that are discussed in Sub-subsection 2.5.1 to give insight about the performance and the programmability of those strategies. Table 4.1 shows the effect of each optimisation applied, on the performance figures, to the simplest coding option for the L100 kernel as described in Section 4.1 and depicted in Figure 4.1.

This option is the most straightforward in terms of the coding required for both the host OpenCL and the FPGA kernel to implement the desired concurrency mapping. In this option the whole of the algorithm's operations are wrapped within a single `for` loop. Such a loop-based kernel is represented in OpenCL as a *Task* kernel. This basic implementation of the kernel is denoted as L100-Seq. This implementation provides the reference execution time for subsequent OpenCL exploration experiments, and is summarised in Table 4.1. The applied clock frequency is 200 MHz which was found to be the maximum that can be used in most cases. Higher frequencies caused failures

to meet timing constraints in the hardware compilation process.

Implementing the L100-Seq version creates an IP block that contains just a single Compute Unit (CU). Since the kernel is implemented as a single task there is, by default, only a single work-item to be mapped to the CU by the host. This can be achieved with a single, simple OpenCL task queue on the host and a single call to enqueue the work-item. Listing 4.1 shows the key steps of the OpenCL Host code to achieve this, along with the outline of the OpenCL Kernel function code.

Listing 4.1: SDSoC OpenCL Initial implementation Host and kernel code fragments for L100 kernel

```
//command queue
cl::CommandQueue q(context, device);
//kernel call
kernel_L100(cl::EnqueueArgs(q,
    cl::NDRange(1, 1, 1),
    cl::NDRange(1, 1, 1)),
    buffer_u, buffer_v, buffer_p,
    buffer_cu, buffer_cv, buffer_z,
    buffer_h, fsdx, fsdy);
//Kernel function
__attribute__((reqd_work_group_size(1,1,1)))
__kernel void L100(
    __global float *u,
    __global float *v,
    __global float *p,
    __global float *cu,
    __global float *cv,
    __global float *z,
    __global float *h,
    const float fsdx,
    const float fsdy
) {
    outer_loop:for (i=0;i<n;i++) {
        inner_loop:for (j=0;j<m;j++) {
            ..    }}
```

The creation of only a single CU is a result of the use of the following statements in Listing 4.1: `cl::NDRange(1, 1, 1)` in the host code and `__attribute__((reqd_work_group_size(1,1,1)))` in the kernel code which specifies that only one work item is to be mapped.

Xilinx provides analysis tools that can be accessed from the SDSoC SDK that support analysis of the L100 kernel performance and latency reports generated during the build process. Table 4.1 shows that the initial implementation requires a latency of 4,714,496 clock cycles (CC) (188.4s in time) to execute. A closer look at the timeline analysis of the kernel's operations² shows that one iteration requires 1,151 CC to finish. These cycles are divided between the kernel's float arithmetic operations, `fmul`, `fadd` and `fsub` and read/write global memory `gmem`, access operations. In the unoptimised initial code the execution of these operations and memory accesses are seen in the timeline analysis to be carried out sequentially (i.e. one operation per clock cycle). Resource usage for the initial implementation, L100-Seq, is given in Figure 4.2.

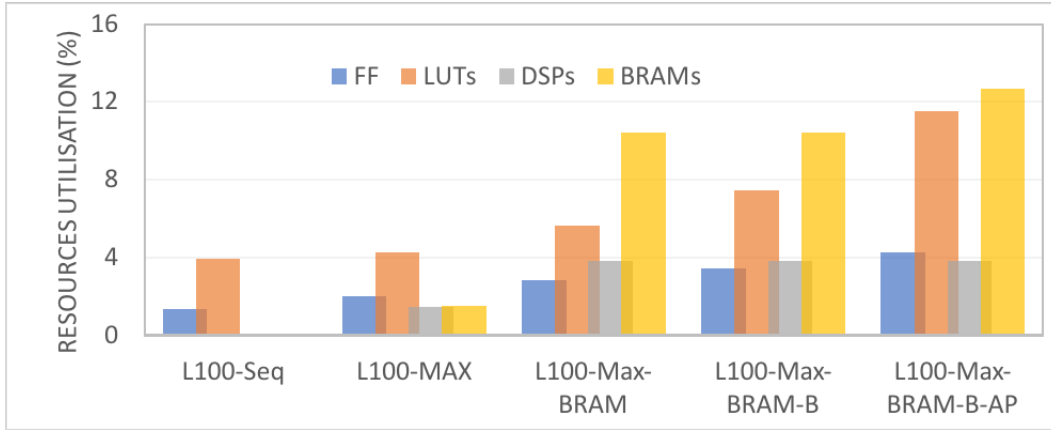


Figure 4.2: Effects of optimisation on resource utilisation.

This straightforward, unoptimised implementation's performance is poor, and the resource usage is low; only a small fraction of the available resources are utilised.

The first optimisation we applied to L100-Seq is the selection of Max Memory Ports resulting in the implementation which is called L100-Max in Table 4.1. Enabling the Max Memory Ports flag in SDx instructs the xocc compiler to utilise the four DDR memory ports, where possible. The compiler reports indicated that this memory port utilisation helped the compiler to pipeline the computational loops with Initiation Interval (II=4) automatically on the inner loop of the kernel's operations. With this optimisation, the kernel latency reduced to 35,457 CC (an 132.96x improvement),

² Available in a report from the build process.

and the iteration latency improved from 1151 CC to 853 CC. The execution time improved by 55.41x. The timeline analysis shows that the use of Max Memory Ports optimises the Global memory (gmem) access operations. Multiple gmem operations are now carried out per CC. The use of Max Memory Ports increased the resource usage compared to the initial implementation, as can be seen in Figure 4.2. More DSPs are utilised, which is consistent with the performance improvement achieved.

The next optimisation, applied on the L100-Max implementation, is utilising the BRAM memory, this resulted in the L100-Max-BRAM implementation in Table 4.1.

Listing 4.2: SDSoc OpenCL L100-MAX-BRAM and L100-MAX-BRAM-B implementations kernel code demonstration

```
// create 2D local memories
float local_u [n_len*n_len];
float local_v [n_len*n_len];
float local_p [n_len*n_len];
float local_cu [n_len*n_len];
float local_cv [n_len*n_len];
float local_z [n_len*n_len];
float local_h [n_len*n_len];

// populate the local BRAM in burst mode
__attribute__((xcl_pipeline_loop(1)))
    for(int i=0; i< n_len*n_len; i++) {
        local_u[i]= u[i];
        local_v[i]=v[i];
        local_p[i]=p[i];
    }
```

In this optimisation, we introduce local BRAM storage for the kernel input data, see Listing ??, then transfer that data from the DDR memory to the BRAMs using a `for` loop. The kernel's operations will access the data from the BRAM buffers, which have much lower data access latency. After the calculations finish, we transfer the data back from the BRAMs to the DDR memory. The timeline analysis shows that the data movement between the BRAMs and the DDR memory take 1404 CC, and the kernel's operations take only 124 CC with `II=1`. With the utilisation of the BRAM memory, the compiler perfectly pipelined the inner-loop of the kernel's operations. This implementation took 2.29 seconds to finish, which improved the execution time 82.27x

compared to the initial implementation. The resource usage for the L100-Max-BRAM implementation is shown in Figure 4.2. The usage of BRAMs increased to 10% compared to 0% in the initial implementation.

To further improve the data-movement from the DDR memory to the local BRAMs, in a further optimisation we used the `burst` mode optimisation strategy. Burst mode is triggered through the use of the pipeline attribute `xcl_pipeline_loop` on the data read/write loops in the kernel. In Table 4.1 this implementation is called L100-Max-BRAM-B. The use of burst mode has improved the performance further and the execution time has decreased to 0.66s. The use of `burst` mode reduced the cost of data movement between the BRAMs and the DDR memory from 1404 CC to 400 CC. That has improved the kernel's latency to 14,805 CC, a 285.48x improvement compared to the initial implementation. The use of `burst` mode has increased only the usage of FFs (3.44%) and LUTs (7.44%) compared to the L100-Max-BRAM implementation.

The final optimisation strategy is the use of array partitioning. The local memory BRAM has only two access ports. Array partitioning thus provides more BRAM access ports for the calculations when the pipeline and unrolling attributes are used. In Table 4.1 this implementation, L100-Max-BRAM-B-AP, shows that no performance improvement compared to the L100-Max-BRAM-B implementation is noticed with the use of array partitioning. The pipeline `II` is already equal to 1, meaning that the BRAM's access ports were already sufficient to allow this initiation interval. In terms of resource usage, the FF, LUTs and BRAM usage increased to FF (4.28%), LUT (11.5%), DSPs (3.80%), and BRAMs (12.7%) compared to the L100-Max-BRAM-B implementation.

4.3.2 Mapping Mechanism Experiments

The following sections presents the results of experiments to exploit the three identified concurrency types (**Instruction-Level-Parallelism**, **Data Parallelism** and **Functional Parallelism**) in the L100 kernel, first, using the mechanisms provided by OpenCL discussed in Section 2.5 and then using the mechanisms provided by Xilinx SDSoC HLS discussed in Subsection 2.5.1. Results in terms of performance and FPGA resource usage are presented and the issues related to programmability discussed. The results in this section build on top of those of the L100-Max-BRAM-B-AP version described earlier in this chapter, so they include the low-level optimisations discussed in the previous section.

4.3.3 Instruction-level parallelism

Instruction-level parallelism can be exploited through the use of the pipeline and unrolling mechanisms for the L100 kernel's operation loops. It should be noted that the use of -O3 compilation flag instructs the xocc compiler to automatically apply the pipeline mechanism to the inner-loop of the kernel's operations. We have seen in section 4.3.1 with the L100-Max-BRAM-B-AP implementation that the compiler has automatically pipelined the inner-loop.

In section 4.1 two loop-based coding options for the kernel were shown. In Figure 4.1(a) a single loop nest contained all the operations, while in Figure 4.1(b), a loop nest was used for each operation.

The SDSoC build reports reveal that the use of the pipeline attribute on the outer loops in both these cases does not affect the total iteration latency. Pipelining the outer loop means that we are trying to pipeline outer iterations that each have 64 inner iterations. The SDSoC compiler failed to translate this design to a register transfer level (RTL) file and the compilation failed.

The use of a loop per operation, as reported for implementation L100-P (b) in Table 4.2, which contains data for each of the implementations discussed in this section, increased the kernel latency to 29,991 CC compared to the L100-P (a) implementation (14,805 CC), and its execution time is 0.97s. The performance differences between implementation L100-P (a) and implementation L100-P (b) come from the fact that for a single loop nest (a) pipelining of the execution of all the kernel's operations is performed. In contrast, with one loop nest per operation (b), the operations are executed in sequence.

Table 4.2: ILP mapping performance of the L100 kernel in SDSoC OpenCL. (P) pipelining, (a) or (b) code options (a) or (b) in Figure 4.1; (U) unrolling.

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Trip Count	Execution Time Seconds	Speedup
L100-P (a)	14,805	524	4096	0.66	285.48x
L100-P (b)	29,991	816	4096	0.97	194.22x
L100-U	15,508	542	1024	0.68	277.05x

The use of unrolling (version L100-U in Table 4.2) provides similar performance figures to the pipeline implementation, L100-P (a). The kernel execution time is 0.68s,

despite the fact that the unrolling mechanism has reduced the trip count of the inner-loop to 1024 iterations. Unrolling results in arithmetic operations from successive iterations being executed in parallel, however, performance can be impacted due to conflicts when accessing the BRAMs. These conflicts may be avoided by increasing the array partitioning factor to provide more BRAM access ports which may be accessed concurrently. The best unrolling factor we found was 4, with an array partitioning factor of 8. L100-U utilised more resources compared with the pipelined version, as can be seen in Figure 4.5, which summarises the resource usage for all the implementations discussed in this section. The design has utilised: FF (6.7%), LUTs (19%), DSPs (3.80%), BRAMs (16.44%).

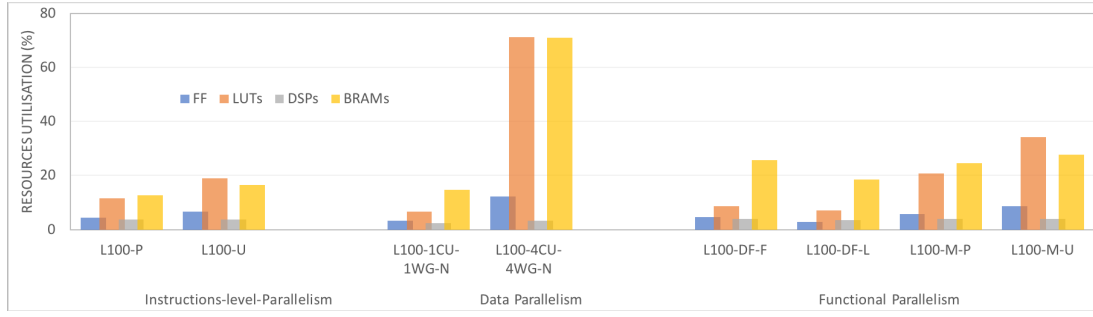


Figure 4.3: Bar graph showing the resource utilisation of the optimised concurrency mapping implementations listed in Section 4.3.1.

In terms of coding the kernel code, we only need to add the pipeline or unrolling attribute to the appropriate loop in the kernel.

4.3.4 Data parallelism

Data parallelism is available in the L100 kernel because the kernel operations are entirely independent as discussed in Section 4.1. The OpenCL NDRange kernel mechanism is suitable for mapping this concurrency type because multiple WIs can be initiated to execute those operations in parallel.

Listing 4.3 shows the coding side of the NDRange mechanism. This coding style is different from that of the Task kernel. There are no loops specified in the kernel for L100, and we use `__attribute__((xcl_pipeline _workitems))` to enable burst-mode and to pipeline the execution of WIs. In the host code, the global and local work size of the kernel is defined using the `cl:NDRange` OpenCL function.

Listing 4.3: NDRange kernel implementation

```

//Host kernel call
kernel_L100( cl::EnqueueArgs(q,
    cl::NDRange(64, 64, 1),
    cl::NDRange(32, 32, 1)),
    buffer_u, ... , fsdx , fsdy);

//NDRange Kernel. 1024 WIs with 4 WGs
__attribute__((reqd_work_group_size(32,32,1)))
//read data in burst mode to Local BRAMs
__attribute__((xcl_pipeline_workitems)) {
loops: l_u [][]=; l_v [][]= l_p [][]=;
}
int j = get_global_id(0);
int i = get_global_id(1);
// WIs Calculate their work
    if(j < N && i < M) {
__attribute__((xcl_pipeline_workitems)) {
        Loops: CU[]=; CV[]=; Z[]=; H[]=;
    }
}
// WIs write data back in burst mode
__attribute__((xcl_pipeline_workitems)) {
    loops: =local_u [][]; =local_v [][];
        =local_p [][];
}}

```

The first, straightforward, NDRange implementation is to create one WG that contains all 4096 WIs which can be executed in parallel. The number of WIs is defined in the host as `cl::NDRange(64, 64, 1)` and in the kernel, the attribute `reqd_work_group_size` is used with the local size of (64, 64, 1). The NDRange function `get_global_id` provides the WI's coordinate within the global work size. The first NDRange design implemented has 64*64 WIs, and is labeled L100-N-1CU-1WG in Table 4.3. This has created a kernel with (4096) WIs and one WG that is mapped to one CU.

This kernel's iteration latency has improved 17.9x (64 CC) compared to the initial implementation (1151 CC). The SDSoc build reports show that 64 read accesses to local memory are carried in parallel in one CC, and the next 64 read accesses happen

Table 4.3: Data parallelism performance of the L100 kernel in SDSoC OpenCL: NDRange. (N) NDRange kernel; (CU) No. of compute units; (WG) No. of work-groups.

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Loop trip Count	Execution Time Seconds	Speedup
L100-N-1CU-1WG	38758	64	4096	1.17	161.02x
L100-N-4CU-4WG	15219	396	1024	1.14	165.4x

in the next CC. In addition, the kernel's arithmetic operations are pipelined with $II = 1$. The use of `__attribute__((xcl_pipeline_workitems))` and multiple WIs has improved the kernel's performance significantly. However, in Table 4.3, the kernel's total latency did not improve to the same extent. This is because data transfers between DDR memory and the local BRAMs add a high overhead. 29,582 CC out of 38,758 CC were needed for data-movement, a result of congestion as many WIs are trying to access the DDR memory through only 4 DDR ports. The resulting execution time of L100-N-1CU-1WG design is 1.17s. For resource usage see Figure 4.5.

In OpenCL NDRange, data parallelism can be exploited at the WI level and also the WG level. The kernel's global data size is divided into multiple local work groups. As the L100 algorithm's operations are completely independent, the WGs can be executed in parallel by mapping each WG to a separate CU. The creation of the WGs is driven by the OpenCL API function `cl:NDRange(32, 32, 1)` in the host where we define the global and the local work size. We specify the number of CUs required through the application project settings in the SDSoC SDK environment.

The implementation L100-N-4CU-4WG in Table 4.3 exploits the data parallelism at the WG level, where the global work size of $64 * 64$ is divided by 4 (by 2 in each dimension), and 4 CUs are requested. The kernel's latency has improved compared to the L100-N-1CU-1WG implementation, and the trip count of the kernel's operations has decreased by 4 (1024). However, no execution time improvement has resulted. The SDSoC build report shows that there are multiple read/write operations to BRAMs carried out in parallel, and the kernel's arithmetic operations are pipelined, but the latency of accessing the DDR memory has degraded the performance. In terms of resources, L100-N-4CU-4WG has created 4 CUs that consume a high number of LUTs and BRAMs, as can be seen in Figure 4.5.

4.3.5 Functional parallelism

Two mapping mechanisms for functional parallelism are available in L100, as discussed in Section 2.5, termed Data-Flow and The number of kernels in that section. These allow each kernel operation to be mapped to a separate compute unit (CU). Data-Flow can be applied over functions or loops. Code options 2 and 3 in Figure 4.1 are suitable for this mechanism. In code option 2, the attribute `xcl_dataflow` can be specified in the kernel and applies to each of the multiple loops, which are then executed in parallel. In code option 3 the same attribute applies to the functions in the kernel.

Table 4.4: Functional parallelism performance of the L100 kernel in SDSoC OpenCL: Dataflow. (DF) dataflow ; (F) apply DF to function code style; (L) apply DF over loops; (M) A kernel per operation; (P) pipeline; (U) unrolling; N NDRange kernel.

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Loop trip Count	Execution Time Seconds	Speedup
L100-DF-F	12,861	-	-	0.60	314x
L100-DF-L	16971	-	-	0.67	281.19x
L100-M-P	56158	88	4096*4	1.62	116.29x
L100-M-U	29974	104	1024*4	1.61	117.01x

Table 4.4 shows the implementations with Data-Flow over functions and loops as L100-DF-F and L100-DF-L, respectively. As can be seen from Figure 4.4, the build reports show that in each implementation 11 CUs have been created by the compiler, 7 CUs for read/write operations and 4 CUs, one for computing each of the operations CU, CV, Z, H. The kernel's execution timeline in SDSoC shows that the kernel is executed in three pipelined steps: load, Compute and Store. In the Compute step the four operations are seen to be carried out in parallel, see Figure 4.4.

The L100-DF-F implementation delivered better performance, with latency 12,861 CC compared to that of the L100-DF-L implementation which had 16,971 CC. L100-DF-F also delivered the best execution time (0.60 seconds) of all the implementations in this exploration study. The resource usage data in Figure 4.5 shows, however, that the L100-DF-F implementation utilised more resources, especially DSPs and BRAMs,

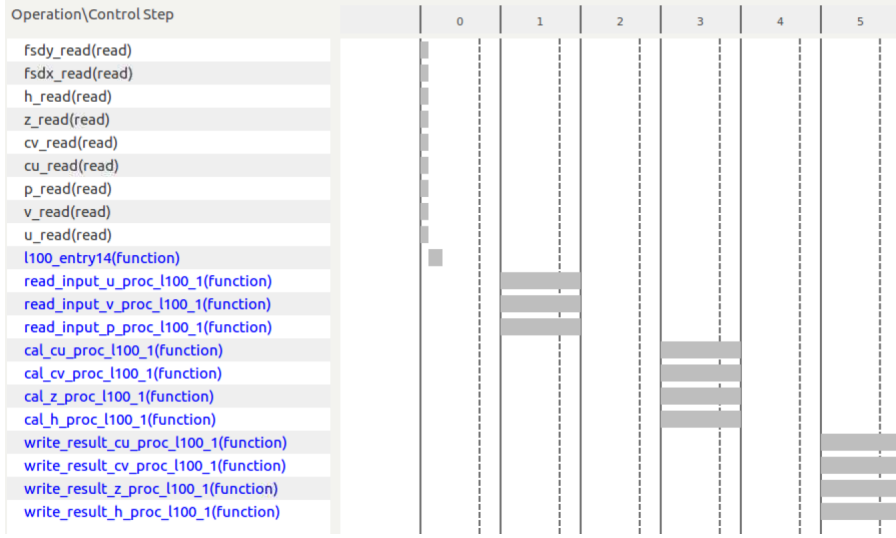


Figure 4.4: Schedule view of the 11 created CUs for L100-DF-F implementation from the SDSoC OpenCL build reports.

than L100-DF-L. The performance differences can be related directly to the resource usage.

Table 4.4 also shows the implementations for the use of Multiple kernels. In this version, we have created a kernel for each of the operations in L100. The kernels can be Task kernels or NDRange kernels.

Given the resource usage figures in the **Data Parallelism** experiments, where one NDRange kernel with 4 WGs mapped to 4 CUs consumed more than 70% of the available LUTs, the use of multiple NDRange kernels is not an option as it will exhaust the FPGA's resources. Therefore, the implementations L100-M-P and L100-M-U use Task kernels. There are four kernels with either the pipeline or unroll attribute applied on the kernel's operation loop. The use of multiple kernels needs synchronisation from the host, where we use an out-of-order command queue to allow parallel execution. This command queue issues the execution of the four kernels in the same clock cycle. The compiler creates 4 CUs and each kernel is mapped to a specific CU. The compiler reports show that the latency is high, however, and execution time is higher in both implementations, (L100-M-P and L100-M-U), compared to the Data-Flow mechanism.

With both Data-Flow and Multiple kernels L100's operations are carried out in parallel. However, with Multiple kernels, we increase the data transfer latency by the number of kernels we create. The data transferred to the local BRAM are for the U, V and P arrays for one kernel, and for four kernels ten data memory transfers are required from the DDR memory. These data transfers are inefficient (compared to

BRAM access) and there are only four DDR memory ports to carry the ten memory access requests. In the Data-Flow mechanism, only three memory data read accesses are carried out in parallel, which explains the performance difference. This overhead has an impact on the latency figures in Table 4.4 for the L100-M-P and L100-M-U implementations. However, in terms of computational iteration latency, L100-M-P and L100-M-U report 88 CC and 104 CC compared to that of the initial implementation with 1151 CC. L100-M-P has lower resource usage than L100-M-U, as shown in Figure 4.5.

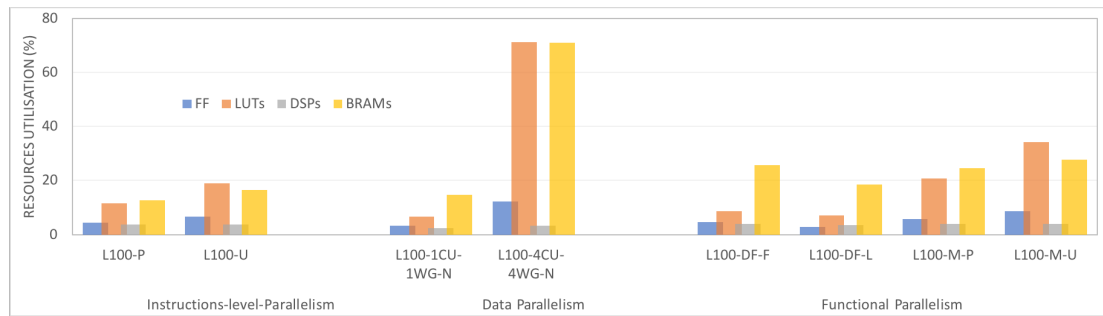


Figure 4.5: Bar graph showing the resource utilisation of the optimised concurrency mapping implementations listed in Section 4.3.2.

4.3.6 Discussion

Multiple mechanisms have been discussed in subsections 4.3.2 to 4.3.5 for mapping the concurrency of the L100 kernel in the SWE shallow water application onto the Xilinx SDSoC FPGA OpenCL programming model. This subsection discusses the performance differences seen, and the sources of inefficiencies, as well as considering programmability issues. Table 4.5 provides some high-level summary information for the discussion.

Performance and Resource Usage Figures: The performance resulting from the use of the different mechanisms was presented mainly in terms of latency and execution time in subsections 4.3.2 to 4.3.5. In addition, we have observed the iteration latency in different implementations. Different ways were explored for mapping the concurrency in the L100 kernel to the target FPGA. The results show that using *Data-Flow over functions*, targeting the functional parallelism in the kernel, provided the best performance in terms of both latency and execution time. Further, this approach utilises fewer FPGA resources compared to the use of multiple kernels or the use of the multiple WG NDRange mechanism.

Table 4.5: Comparison of the number of compute units (CUs), the number of command queues (CQs), and loops, and the coding difficulty (score from 1 to 5) and resource usage of different mechanisms implemented on the L100 kernel in SDSoC OpenCL.

Task Kernel	NDRange Kernel	DF Functions	DF Loops	Multiple Kernels
1 CU	4 CUs	13 CUs	13 CUs	4 CUs
1 in-order-CQ	1 in-order-CQ	1 in-order-CQ	1 in-order-CQ	1 out-of-order-CQ
1 Loop	No Loop	No Loop	11 Loops	12 Loops
2	5	4	4	4
Low Resource Usage	High Resource Usage	Low Resource Usage	Low Resource Usage	Medium Resource Usage

Code option 3 in Figure 4.1 proved to be the better of the coding choices to map the L100 concurrency. Since the kernel’s operations are independent, processing them all in parallel seems to be the appropriate method to achieve good computational performance — for L100-DF-F, a latency of just under 13,000 CC and execution time of 0.60s, with a speedup of 314x over the initial version. The Data-Flow over functions mechanism assigned each operation in the kernel to its own CU and executed them in parallel after filling the local BRAMs with the required input data. Exploiting instruction level parallelism with the pipeline mechanism (L100-P (a)) came second, with nearly 2000 CC extra latency, execution time of 0.66s and speedup 285.48x over the initial version. This version had similar resource usage, except that the Data-Flow implementation used more BRAMs.

In terms of iteration latency (a measure of the level of data parallelism), the use of the NDRange mechanism with 1CU and 1WG provided a speedup of 17.9x. This implies a potential to exploit a high level of parallelism, since 4096 WIs are set to be executed in parallel. However, the pressure on DDR memory bandwidth to support 4096 WIs severely degrades the performance in this case.

In terms of resource usage, Table 4.5 and Figure 4.5 show that the *multiple WGs NDRange* kernel and the *multiple kernels* implementation consumed the highest percentage of resources. Utilising a large amount of resources does not guarantee high performance, mainly due to memory access overheads in this case. However, use of fewer resources can be expected to lead to lower power use, though we did not pursue this here.

Mapping Mechanisms Programmability: Quantifying *programmability* is a challenging task. Therefore, in this subsection, we choose to observe what code changes were required to the original initial implementation and give some qualitative indication of the programming *difficulty* involved in our implementations. Programming FPGAs requires traditional scientific programmers to take on a new computational model involving the low-level design of hardware as well as often novel high(er)-level programming models for FPGA systems. For example, OpenCL introduce new design choices and trade-offs that become easier to deal with with experience. Table 4.5 summarises some high-level aspects of those choices and trade-offs³. The NDRange kernel mechanism with multiple WGs was the largest and most complex to design and implement. The implementation required numerous changes to the initial code, and the exploration of various designs to find the best solution given the use of local BRAMs. In contrast, the use of the pipeline and unroll mechanisms were the most straightforward to apply since the changes to the initial implementation required only inserting SDSoc attributes on the loops.

The exploitation of Functional parallelism with the Data-Flow and multiple kernel mechanisms was in the middle in terms of coding difficulty. Applying Data-Flow over functions required coding in the functional style, while the Data-Flow over loops required only the insertion of an attribute in the kernel top-level function. Both Data-Flow mechanisms needed changes to the number of local BRAMs used (achieved with local data declarations) and, therefore, extra data copies to read to and write from the BRAMs. The Multiple kernels mechanism required a considerable amount of time to code the four kernels. In addition, changes to the host code were required to be explored and implemented, as indicated in Table 4.5.

4.4 L100 kernel mapping using Vivado HLS

The use of the Vivado HLS tool approach involved two paths to map the L100 kernel's concurrency to the ZCU102 FPGA. As described in the Section 2.4 in the background Chapter 2, the first path is through the use of Vivado HLS tool (Compile Stage), which involves producing an FPGA IP block for the L100 kernel. In this stage, there are different HLS pragmas that can be used and the various L100 kernel coding options that are available can affect mapping the L100's concurrency levels. The second path is

³The scoring judgement in Table 4.5 is being made by the thesis author who is more from a software development background but familiar with FPGAs, and this likely matches the background of potential users.

through the use of the Vivado design suite (Link Stage), where there are plenty of low-level hardware design options that can be explored in order to form a system design which includes the L100 IP block. The hardware design decisions involve choices such as selecting appropriate communication blocks, data addresses management, designing the memory solutions (e.g. the use of DDR memory and/or BRAM memory), the selection of connection ports and the data path parameters, including sizes and data widths.

4.4.1 Vivado Initial Implementation

The first exploration that is carried out, in mapping the L100 concurrency using the Vivado approach, explores the initial implementation that represents the base code for the mapping mechanism implementations. In addition, it explores the impact on performance and resource usage of applying the low-level FPGA optimisations that are discussed in section 2.5.1. In conducting this initial exploration, two steps were carried out first: The first step is building the L100 kernel IP block, and the second is creating the system design that hosts the L100 kernel IP block. Listing ?? shows the Vivado HLS kernel code for building the L100 kernel IP block. The HLS pragma `HLS INTERFACE` is used in the kernel code to define data depth and data burst attributes for each kernel argument. In the base code, all the kernel's arguments are bundled to one port that can connect to the memory block. Two available memory solutions (DDR memory and BRAM blocks) are available for storing the shared data between the Host (ARM CPU) and the L100 IP block. The use of DDR memory was already explored in the L100 implementation with OpenCL, see Section 4.3. Therefore, the memory solution choice for the L100 Vivado exploration is to explore the use the BRAM block memory. To enable the access of the L100 IP block argument ports to the BRAM block we bundled all the ports to `bram` in the `HLS INTERFACE HLS` pragma, as shown in Listing ?. The L100 kernel code in Listing ?? is compiled using the Vivado HLS tool which results in an IP block for L100 kernel, see Figure 4.6. The L100 IP block has two ports: `m_axi_bram` for accessing the BRAM storage, and `s_axi_AXILiteS` for the ARM CPU to control the execution of the kernel.

Listing 4.4: Vivado Initial implementation kernel code structure for L100 kernel

```
// Kernel function
int L100_seq(
float *u,
```

```

float *v,
float *p,
float *cu,
float *cv,
float *z,
float *h
) {

#pragma HLS INTERFACE m_axi depth=128
    port=u offset=slave bundle=bram \
    num_read_outstanding=8 num_write_outstanding=8 \
    max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
    port=v offset=slave bundle=bram \
    num_read_outstanding=8 num_write_outstanding=8 \
    max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
    port=p offset=slave bundle=bram \
    num_read_outstanding=8 num_write_outstanding=8 \
    max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
    port=cu offset=slave bundle=bram \
    num_read_outstanding=8 num_write_outstanding=8 \
    max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
    port=cv offset=slave bundle=bram \
    num_read_outstanding=8 num_write_outstanding=8 \
    max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
    port=z offset=slave bundle=bram \
    num_read_outstanding=8 num_write_outstanding=8 \
    max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
    port=h offset=slave bundle=bram \
    num_read_outstanding=8 num_write_outstanding=8 \

```

```

max_read_burst_length=32 max_write_burst_length=32

#pragma HLS INTERFACE s_axilite port=return
outer_loop:for (i=0;i<local_n;i++) {
    inner_loop:for (j=0;j<local_n;j++) {
        .. } // operation code
    }
}

return 0;

```

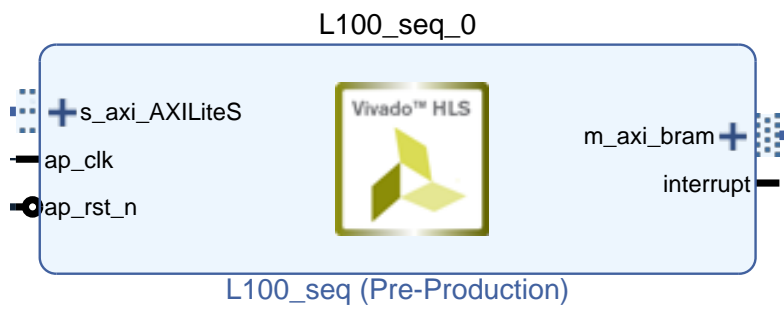


Figure 4.6: Vivado L100 kernel IP block.

The second step is creating the system design using the Vivado design suite to host the L100 kernel IP block shown in Figure 4.6. Figure 4.7 shows the created system design. This system design is inspired by the design proposed in [ARAM19], which will be further discussed in Chapter 7. The system design consists of:

- One AXI BRAM Controller and One Block Memory Generator to provide the BRAM memory solution for the L100 IP block.
- The number of AXI Crossbars to be used for the data-path conversion, where necessary.
- One AXI Protocol Converter to convert between the AXI4 port in the ZynQ block and the AXI_Lite port in the L100 IP block.
- One AXI Interconnect for converting the 128 bit data path (of the ZynQ block port) to the 32 bit (L100 IP block port).
- One Clock Wizard block that provides multiple clock frequencies that can be used by the system design block and the L100 IP block.



Table 4.6: The performance figures of each optimisation on the L100 kernel in Vivado HLS. (Seq) no optimisations; (BRAMs) use of BRAM memories; (B) use burst mode; (AP) use array partitioning

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Loop Trip Count	Execution Time Seconds	Speedup
L100-Seq	1290369	20162	64	15.54	-
L100-BRAM	470042	5940	4225	5.27	2.94x
L100-BRAM-B	402496	5938	4225	3.59	4.32x
L100-BRAM-B-AP	1770564	27229	4225	15.75	-1.01x

Table 4.6 shows the performance effect of applying the low-level FPGA optimisations, in the order that is discussed in section 4.2, when applied to the simplest coding option, (a), for the L100 kernel. In this option, the whole of the algorithm's operation code is wrapped within a single `for` loop. This basic implementation of the kernel is denoted as L100-Seq (as we did in the OpenCL exploration, see 4.3). L100-Seq represents the initial execution time for the subsequent Vivado exploration experiments, and is summarised in Table 4.6. The applied clock frequency is 447 MHz, which was the maximum that could be used in most cases. Higher frequencies caused failures to meet timing constraints in the hardware compilation process. The Vivado L100 design clock frequency is much higher than the SDSoC OpenCL L100 version which is likely due to the added complexity imposed by the SDSoC OpenCL tooling.

The L100-Seq IP block is generated and inserted into the system design in Figure 4.7. Then this design is compiled to generate the bitstream files. Table 4.6 shows that the initial implementation requires a latency of 1290369 clock cycles (CC) (15.54s in time) to execute. The Vivado HLS tool provides a timeline analysis report that shows that one iteration requires 20162 CC to finish. These cycles are divided between the kernel's float arithmetic operations and accessing the data from the external BRAM block in Figure 4.7. Resource usage is given in Table 4.7. This straightforward, unoptimised implementation's performance is poor, and the resource usage is low; only a small fraction of the available resources are utilised.

Table 4.7: The resource usage figures of each optimisation on the L100 kernel in Vivado HLS. (Seq) no optimisations; (BRAMs) use of BRAM memories; (B) use burst mode; (AP) use array partitioning.

Implementation Name	FF	LUT	DSP	BRAM_36K
L100-Seq	7907 (1%)	7054 (2%)	16 (0%)	2 (0%)
L100-BRAM	7918 (1%)	6183 (2%)	16 (0%)	79 (4%)
L100-BRAM-B	8033 (1%)	6469 (2%)	16 (0%)	79 (4%)
L100-BRAM-B-AP	10425 (1%)	9209 (3%)	17 (0%)	114 (6%)

To improve the Vivado L100 kernel performance, we started with the order of optimisations mentioned section 4.2. Any change to the Vivado L100 kernel requires re-generating of the L100 IP block, re-inserting it to the system design and re-generating the bitstream files. The Max Memory ports optimisation is not suitable for the Vivado design because we use the BRAM block, not the DDR memory. The BRAM block only provides two memory ports, one used for reading access and writing access. Therefore, the first optimisation we applied to the L100-Seq IP block is utilising the BRAM memory. This involves utilising local BRAM memory within the L100-Seq IP block; this is L100-BRAM in Table 4.6. The local BRAM storage is introduced for the kernel input data. That data is then transferred from the external BRAM block in the system design, see Figure 4.7. The kernel's operations will access the data from the local BRAM buffers, which have much lower data access latency as they are inside the L100 kernel. After the kernel's calculations finish, we transfer the data from the local BRAM buffers to the external BRAM block storage. The timeline analysis from Vivado shows that the kernel's latency and iteration latency has improved by 2.74x and 3.39, respectively, compared to the initial implementation, as can be seen in Table 4.6. In addition, this implementation took 5.27s, which improved the execution time by 2.94x compared to the initial implementation. The resource usage for the L100-BRAM implementation is shown in Table 4.7. The key figure is the BRAM usage which increased to 4% compared to 0% in the initial implementation.

To improve the data movement from the external BRAM memory to the local

BRAM blocks, we used the `burst` mode optimisation strategy. Burst mode is triggered through the use of the C function `memcpy`. The use of burst mode has improved the performance further and the execution time decreased to 3.59s, see experiment L100-BRAM-B in Table 4.6. In addition, this optimisation has improved the kernel's latency to 402496 CC from 1290369 CC in the initial implementation. The use of burst mode has slightly increased only the FFs and the LUTs usage figures compared to the L100-BRAM implementation.

The final optimisation strategy, as mentioned in the optimisations order in section 4.2 is the use of array partitioning. Array portioning increases the local BRAM bandwidth by providing more BRAM access ports. In Table 4.6 this implementation, L100-BRAM-B-AP, shows a performance degradation (15.75s) compared to the initial implementation (15.54s). In the L100 Vivado exploration we replaced the DDR memory used in the SDSoC OpenCL exploration (4.3) with an external BRAM storage block. The memory path between the local (partitioned) BRAM blocks and the external BRAM memory (2 memory ports) is narrower compared to that of the DDR memory (4 memory ports). This has led to limited bandwidth where the use of array partitioning increased the number of memory access required in filling those local BRAMs from the external memory. This issue will be investigated more in the functional parallelism exploration in 4.4.4.

The Vivado HLS reports show that II of the four different versions in Table 4.6 are null; Which means that the Vivado compiler did not undertake any automatic pipelining for the loops. In contrast to the SDSoC OpenCL, the `xocc` compiler has undertaken some automatic optimization (pipeline) as illustrated in section 4.3.

4.4.2 Mapping Mechanism Experiments

This section presents the results of the exploration experiments carried out on the Vivado HLS approach to map only two of the L100 kernel's identified concurrency types (**Instruction-Level-Parallelism** and **Functional Parallelism**). Data parallelism depends on the availability of a mechanism to allow the processing of the L100 kernel's iterations in each loop in parallel. This was feasible to map in the SDSoC OpenCL approach through the use of NDRange kernels. There is no mechanism to support the equivalent in Vivado HLS in order to map this concurrency type. The Vivado HLS mechanisms that are available, which are discussed in section 2.5.1, are utilised for these mapping experiments. Results in terms of performance, resource usage, and

programmability issues are discussed and presented in the following. The implementations in this section are built on top of the L100-BRAM-B version in Table 4.6, so they include the low-level optimisations discussed in the previous section.

4.4.3 Instruction-level parallelism

Instruction level parallelism is available through the use of the HLS pipeline pragma mechanism for the operation loops in the L100 kernel. In the SDSoC OpenCL exploration, we found that the performance of the mappings when using a loop per operation (L100-P (b)) and unrolling (L100-U) was lower than when using the pipelining (L100-P (a)) implementation, see section 4.3.3. Thus, motivated by the results of the OpenCL explorations, we only applied the implementation of version L100-P (a) in this concurrency type exploration to avoid repetition of unnecessary experiments.

The pipeline pragma is applied to the code option (a) in Figure 4.1, where a single loop nest contains all the operations. Table 4.8 shows that the L100-P (a) implementation improved the L100 performance by 58.86x compared to the initial implementation. This performance improvement is a result of the inner loop operations being pipelined with $\Pi=2$. The kernel latency is reduced to 37915 CC, which helped achieve 0.264s execution time. Table 4.9 shows increases in resource usage compared to the initial implementation, especially in BRAM usage (9%).

4.4.4 Functional Parallelism

Two mapping mechanisms for Functional parallelism are available in the Vivado HLS approach. As described in Section 2.5, they are: Data-Flow and the number of kernels. Motivated by the results of the SDSoC OpenCL explorations in 4.3.5, we only applied the Data-Flow over functions with regard to the use of Data-Flow mechanisms in the Vivado HLS exploration since Data-Flow over functions showed better performance in the OpenCL results compared to the performance of Data-Flow over loops version.

Table 4.8 shows two Vivado Data-Flow implementations which are L100-DF-F-1-BRAM Block and L100-DF-F-7-BRAM Blocks. The number of BRAM blocks in the implementation's name is the number of external BRAM blocks utilized in the system design. Both implementations have the same kernel code, which is shown in Listing 4.5; Therefore, the same L100 IP block is generated for the both implementations, see Figure 4.8. The kernel's operations in this implementation are following code option (C)

Table 4.8: The performance figures of the ILP and Functional parallelism mapping of the L100 kernel in the Vivado approach. (P) pipelining; (a) code option (a) in Figure 4.1; (DF) dataflow, (F) apply DF to function code style; (M) A kernel per operation. The blank entries in this table mean that the information is not available to be reported. The speed up is relative to the Vivado L100-seq initial implementation.

Concurrence Type	Experiment Name	Latency (Clock Cycles)	Iteration Latency	Loop Trip Count	Execution Time Seconds	Speedup
Instruction-Level Parallelism	L100-P-a	37915	114	4225	0.264	58.86x
Functional Parallelism	L100-DF-F 1-BRAM Block	16740	-	-	0.322	48.26x
	L100-DF-F 7-BRAM Blocks	16740	-	-	0.164	94.75x
	L100-M-P-4-BRAMS Blocks	-	-	-	0.238	65.29x

in Figure 4.1 of mapping the kernel's operations to four functions. In the implementation L100-BRAM-B, which we build these mapping mechanism experiments on top of, we bundled all the kernel's argument ports to one `bram` port. That means the L100 IP block would have only one memory port connected to the external BRAM block for the read/write access operations. In the L100-DF kernel code, we designed the kernel to have three memory read functions that are executed in parallel and four memory write functions that also executed in parallel, see Listing 4.5. This kernel design decision was explored in the SDSoC OpenCL exploration and showed a high performance. In the SDSoC OpenCL operations we found that the parallel access to the DDR memory improved the data-movement which, in turn, improved the overall kernel's execution time. Enabling parallel read/write was available using the `max memory ports` mechanism, which ensures that each kernel argument has a separate memory port. To enable the parallel read/write operations in the Vivado Data-Flow implementation, we bundled L100 kernel arguments to separate `bram` ports which allows parallel execution of the read/write functions, see code Listing 4.5. The `bram` ports are distinguished with port numbers from 0 to 6, as shown in the code Listing 4.5.

Listing 4.5: Vivado L100-DF implementation kernel code for L100 kernel

Table 4.9: The resource usage figures of the ILP and Functional parallelism mapping of the L100 kernel in the Vivado approach. (P) pipelining; (a) code option (a) in Figure 4.1; (DF) dataflow, (F) apply DF to function code style; (M) A kernel per operation.

Concurrency Type	Experiment Name	FF	LUT	DSP	Total Local & External BRAM 36K Usage
Instruction-Level Parallelism	L100-P-a	13327 (2%)	9028 (3%)	39 (1%)	83 (9%)
Functional Parallelism	L100-DF-F 1-BRAM Block	21124 (3%)	16791 (6%)	65 (2%)	259.50 (28%)
	L100-DF-F 7-BRAM Blocks	21124 (3%)	16791 (6%)	65 (2%)	283.50 (31.09%)
	L100-M-P-4-BRAMS Blocks	21077 (3%)	15962 (5.85%)	65 (2%)	195.50 (21%)

```
// Kernel function
int L100_df(
float *u,
float *v,
float *p,
float *cu,
float *cv,
float *z,
float *h
) {

#pragma HLS INTERFACE m_axi depth=128
port=u offset=slave bundle=bram0 \
    num_read_outstanding=8 num_write_outstanding=8 \
    max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
port=v offset=slave bundle=bram1 \
```

```

        num_read_outstanding=8 num_write_outstanding=8 \
        max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
port=p offset=slave bundle=bram2 \
        num_read_outstanding=8 num_write_outstanding=8 \
        max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
port=cu offset=slave bundle=bram3 \
        num_read_outstanding=8 num_write_outstanding=8 \
        max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
port=cv offset=slave bundle=bram4 \
        num_read_outstanding=8 num_write_outstanding=8 \
        max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
port=z offset=slave bundle=bram5 \
        num_read_outstanding=8 num_write_outstanding=8 \
        max_read_burst_length=32 max_write_burst_length=32
#pragma HLS INTERFACE m_axi depth=128
port=h offset=slave bundle=bram6 \
        num_read_outstanding=8 num_write_outstanding=8 \
        max_read_burst_length=32 max_write_burst_length=32

#pragma HLS DATAFLOW

//Data Read from External BRAM
//Read u,v and P
read_u(u, local_u)
read_v(v, local_v)
read_p(p, local_p)

//Compute functions
//CU,CV,Z and H
Compute:
local_cu (local_u, local_p);

```

```

local_cv (local_v , local_p );
local_z  (local_u , local_v , local_p );
local_h  (local_u , local_v , local_p );

//Data Write back to External BRAM
// Write CU,CV, Z and H
write_cu(local_cu , cu)
write_cv(local_cv , cv)
write_z (local_z , z)
write_h (local_h , h)

...

// ..... end .....

```

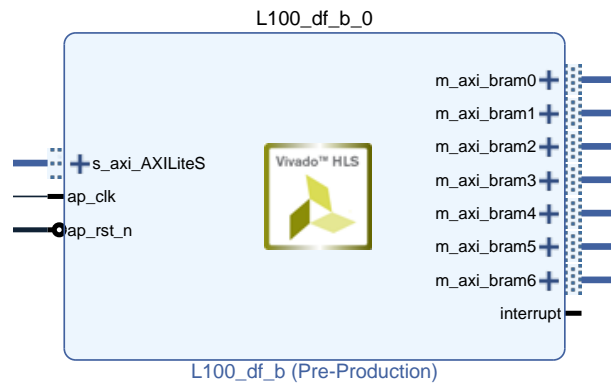


Figure 4.8: Vivado L100-DF IP Block.

As the L100-DF IP block created has seven ports, the parallel read/write accesses to the external BRAM also required multiple ports in the external BRAM block. However, the external BRAM only has one data access port. This would cause bandwidth congestion and overhead in the implementation's execution time. To explore this issue, the Vivado Data-Flow version named L100-DF-F-1-BRAM-Block in Table 4.8 was created with just one external BRAM block that has only one data access port, as in Figure 4.9. The resulting performance compared to the L100-P-a version is slower with 0.322s execution time. The L100-DF-F-1-BRAM-Block implementation thus demonstrated the need, and benefit, for more memory bandwidth to efficiently map the L100 kernel's functional parallelism through the Data-Flow mechanism. To

improve the memory bandwidth, we explored the memory design solution of creating seven external BRAM blocks. Thus, a BRAM memory per argument is created in this design where each memory holds one array (u , v , P , CU , CV , Z and H), see Figure 4.10. Table 4.8 shows that this memory design solution (L100-DF-F-7-BRAM-Blocks) has improved the performance by 94.75x compared to the initial implementation. The utilisation of seven external BRAM blocks improved the kernel's execution time from 0.322s (with one external BRAM block) to 0.164s. In addition, the kernel latency has improved to 16740 CC compared to 1290369 CC of the initial implementation. Table 4.9 shows increases in resource usage of FF (3%), LUT (6%) and DSP (2%) compared to the initial implementation. The BRAM usage was the highest increase figure as we utilised (31.09%) of the BRAM resources.

Table 4.8 also shows results for the implementation (L100-M-P-4-BRAM-Blocks) which was designed to explore the use of the multiple kernels mapping mechanism. In this version, we created a kernel for each of the operations in the L100 kernel. Motivated by the SDSoc OpenCL exploration results, the chosen mapping mechanism for each of the four kernels' operations is the pipeline pragma. The four kernel codes are following code option (a) in Figure 4.1, with the pipeline pragma applied to the inner loop in each kernel. Using the Vivado HLS tool, four IP blocks are generated which compute CU , CV , Z and H , respectively. Figure 4.11 shows the system design for this L100-M-P-4-BRAM-Blocks implementation. Each operation kernel is connected to its own external BRAM block, so this implementation has four external BRAM blocks. Input arrays u , v and P are shared between the four kernels. In order to enable each IP block access to those arrays we duplicate them in the external BRAM blocks in order to avoid conflict for bandwidth. This implementation delivered an execution time of 0.238s and speedup of 65.29x compared to the initial implementation, see Table 4.8. Similarly to the other mapping mechanisms, the resource usage of this implementation increased, to FF (3%), LUT (5.85%), DSP (2%) and BRAM (11%) compared to the initial implementation, see Table 4.9.

4.4.5 Discussion

Multiple Vivado HLS mechanisms have been utilised for mapping the L100 kernel's concurrency. This section discusses the performance differences seen, the sources of inefficiencies found and considers programmability issues.

Performance and Resource Usage Figures: The performance of the different



Figure 4.9: Vivado L100-DF-F-1-BRAM-Block implementation system design.

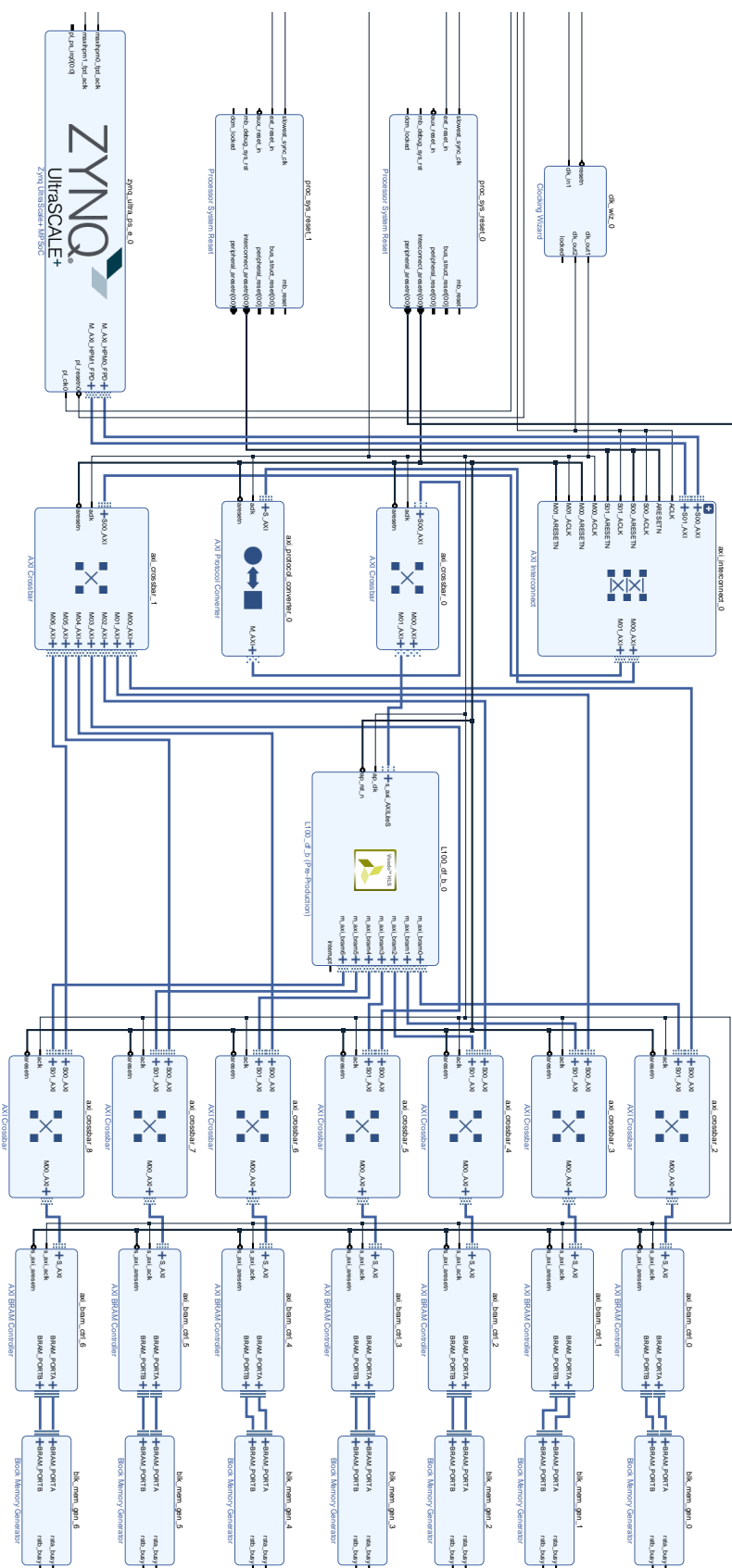


Figure 4.10: Vivado L100-DF-F-7-BRAM-Blocks implementation system design

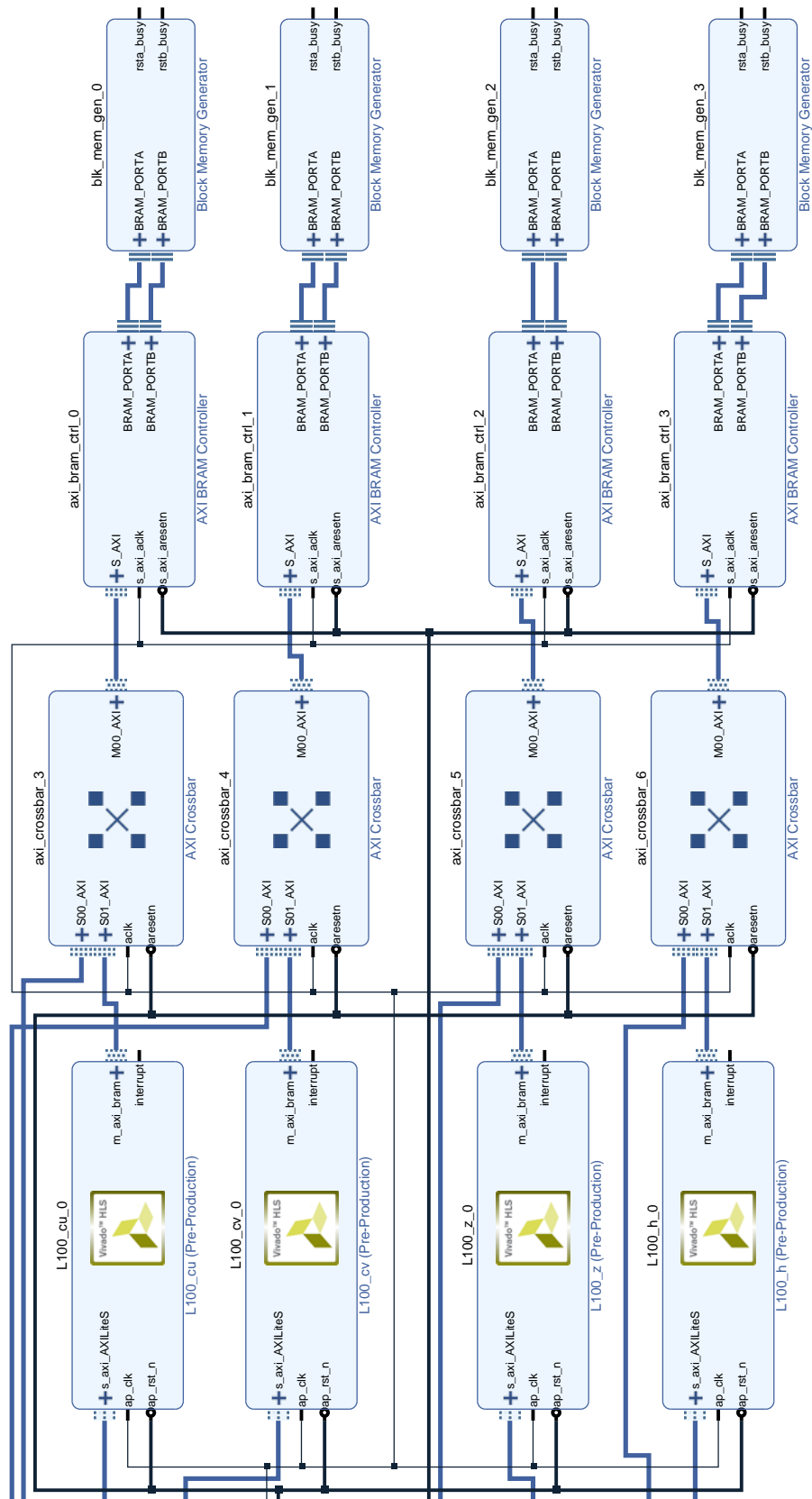


Figure 4.11: Vivado L100-M-P-4-BRAM-Blocks implementation system design.

mechanisms was presented in terms of latency and execution time. Moreover, the iteration latency was reported in different implementations, where available. The results show that using `Data-Flow` over functions with seven external BRAM blocks, targeting the functional parallelism in the L100 kernel, provided the best performance in terms of both latency and execution time (L100-DF-F-7-BRAM-Blocks). Code option (C) in Figure 4.1 proved to be the better coding option to map the L100 kernel's concurrency. Processing the kernel's operations in parallel, where possible, is the appropriate method to achieve good computational performance since the kernel's operations are independent. The other functional parallelism implementation of utilising the multiple kernels mechanisms (L100-M-P-4-BRAM-Blocks) came second with an execution time of 0.238s and speedup 65.29x, with less usage of LUT and BRAM resources. This is a good example of the trade-offs available to the programmer between performance and resource usage.

In terms of resource usage, Table 4.9 shows that the use of `Data-Flow` mechanisms consumed more resources compared to the other implementations, especially the use of BRAM blocks.

Mapping Mechanisms: Programmability: We have observed the level of programming difficulty in each implementation in terms of the host and the kernel code, as well as the system design changes that were required to be made to the original initial implementation. In addition, we give some qualitative indication of the programming difficulty involved in our implementations, which is summarised in Table 4.10⁴.

Programming FPGAs using the Vivado HLS approach requires detailed understanding and low-level hardware design knowledge of both the FPGA hardware and Vivado programming mechanisms, such as those associated with the memory solution design and implementation and data addresses management for connecting BRAM memories appropriately. Table 4.10 summarises the programming difficulty level involved with each implementation.

The easiest implementation to carry out was L100-P-a. The only change that was required to the baseline implementation was applying the `pipeline HLS` pragam over the inner loop of the kernel's operations. In addition, inserting only the single IP block created for L100-P-a to the system design was required. This implementation required a similar amount of FPGA resources compared to the other implementations.

Both `Data-Flow` implementations required many changes to the kernel code and

⁴The scoring judgement in Table 4.10 is being made by the thesis author who is more from a software development background but familiar with FPGAs, and this likely matches the background of potential users.

these were complex changes compared to the original initial implementation. Coding the host code and creating the system design for the Data-Flow implementation with one external BRAM block was straightforward and did not require any changes other than inserting the IP block to the system design (straightforward but not trivial). However, the Data-Flow implementation with seven external BRAM blocks required a great deal of host code and system design changes. The changes in both the host code and the system design were complex and needed high coding effort. This implementation required the highest amount of BRAM resources.

The final implementation involving multiple kernels was very complex. The kernel code required multiple changes as we had to code four kernels and generate four IP blocks. The host code was also amended to manage four kernels with their four external BRAM blocks, instead of managing only one kernel as in the original initial implementation. Creating the system design was challenging in order to accommodate four IP blocks and manage the connection of four external BRAM blocks and their address management. This implementation required a higher amount of resources compared to the original initial implementation.

Table 4.10: Comparison of the programming difficulty levels (score from 1 to 5) between the different mechanisms implemented on the L100 kernel in the Vivado HLS approach.

Experiment Name	Host Code Difficulty	Kernel Code Difficulty	System Design Difficulty	Resource Usage
L100-P-a	2	2	2	Low
L100-DF-F 1-BRAM Block	2	5	2	High BRAM Usage
L100-DF-F 7-BRAM Blocks	4	5	4	Very High BRAM Usage
L100-M-P- 4-BRAMS Blocks	5	5	4	High BRAM Usage

4.5 Summary

The main aim of this chapter was to explore, from the viewpoint of the traditional scientific HPC software developer, the wide range and levels of approaches and mechanisms available for exploiting FPGAs in scientific applications. The exploration highlighted numerous trade-offs involved in the design and implementation phases and demonstrated their impact on performance, resource usage, and programmability to a less quantitative extent. This exploration aimed to provide traditional scientific programmers with insight into how best to exploit FPGAs in their applications. This chapter explored the mechanisms available to scientific programmers in a relatively high-level HLS approach, SDSoC OpenCL, and a low-level HLS approach, Vivado, to map the concurrency types available within a single kernel, L100, in both HLS approaches.

The results of this exploration study can be summarised as follows. The Data-Flow over functions mechanism can extract the best performance out of a single kernel that has multiple independent operations. In both HLS approaches (SDSoC OpenCL and Vivado) implementations targeting the functional parallelism in the L100 kernel, with the Code option (C) in Figure 4.1 proved to be the better coding option to map the L100 kernel's concurrency. Processing the kernel's operations in parallel, where possible, is the appropriate method to achieve good computational performance since the kernel's operations are independent. The Data-Flow over functions implementation assigned each of the kernel's computational operations to its own compute unit and launched them in parallel from the host.

However, this method can have high FPGA resource usage and requires a relatively high programming effort.

Chapter 5

Exploratory Study (1) Part Two: SWM Multiple-Kernels Mapping

This chapter presents the second part of the *exploratory study* (1) for mapping and optimizing the whole SWM application kernels and exploring the kernel-to-kernel communication implementation options using SDSoC OpenCL and Vivado HLS tools. This chapter shows the application of the learned lessons in Chapter 4 for mapping the L100 kernel concurrency type to optimize the other SWM kernels. Two main sections are presented in this chapter, Section 5.1 and Section 5.2. Section 5.1 presents the mapping of the SWM application using the SDSoC OpenCL, and Section 5.2 presents the use of the Vivado approach for mapping the SWM multiple kernels. In each section, we first explore the suitable mapping mechanism to optimize each of the SWM kernels and explore their impact on performance and resource usage for a fixed problem size. Following that, we identify the maximum possible size of the problem based on the applied optimisations and the finding number of kernels possible to implement out of the nine SWM kernels. Moreover, the final step in these two sections explores options for managing the data movement associated with communication between the SWM kernels. The chapter ends with a section summarizing the main lessons learned from the experiments. The performance results in this exploration chapter are the kernel compute time only and not overall runtime (which will be reported in Chapter 6.)

5.1 SWM Kernels Mapping Using SDSoC OpenCL

5.1.1 Optimise the SWM SDSoC OpenCL kernels

The L100 concurrency mapping exploration with the SDSoC OpenCL in Section 4.3 from Chapter 4 revealed that the Data-Flow mechanism was the most suitable option for mapping the kernel's concurrency. The identified concurrency types in Section 4.1 from Chapter 4 for the L100 kernel are examples of the similar concurrency types in other kernels in the SWM application. The SWM's kernels that share similar concurrency types to the L100 kernel are the L200 and L300 kernels. As mentioned previously in Section 2.6.1 from Chapter 2, L100, L200 and L300 are computationally intense kernels, since they execute large loops numerically-intense iterations with independent operations.

We have, therefore, applied the Data-Flow over functions mechanism targeting the functional parallelism in the L200 and L300 kernels. Listing 5.1 and 5.2 show the SDSoC OpenCL kernel codes for the L200 and L300 kernels, respectively. Note that the Data-Flow mechanism was applied on top of the same key-optimisations that we explored in the L100 SDSoC OpenCL initial implementation, as discussed in Section 4.3.1 from Chapter 4.

Following this, we optimised the halo kernels (L100pc and L200pc) with the use of a pipeline mechanism similar to the L100-P-a implementation in the SDSoC OpenCL exploration study in Section 4.3 from Chapter 4. The reason for choosing this mapping mechanism is that the halo kernels do not include functional parallelism.

The SWM initialisation kernels `init1`, `init2` and `L300pc` are kernels that are executed only one time during the lifetime of the SWM application. Therefore, there is the option to either implement them with optimisations method that do not consume high amount of resources or call them only on the host to as it is worth sparing some FPGA resources for the kernels called multiple times. We have tested optimising these one-time initialisation kernels with the pipeline mechanism similar to the L100-P-a implementation in Section 4.3 Chapter 4, and compile all the SWM nine (compiler flag applied was -O3, and the clock frequency 200MhZ which was the max possible). This test shows that it was not possible to have all nine optimised SWM kernels compiled even when choosing very small problem sizes such as $(2*2)$,¹. Having all the nine optimised kernels required more resources than the ZCU102 FPGA can provide. To

¹The compilation process always fails if one or more of the FPGA resources (FF, LUT, DSP, BRAM) utilisation percentage exceeded 80%.

help reduce the resource usage and allow compilation of a bigger sized problem than $(2*2)$, we chose not to compile for the FPGA the kernels that are only called one time from the host CPU code, such as the `init1`, `init2` and `l300pc` Kernels. These three kernels can be executed in the host ARM CPU instead. This design decision can spare more resources for the main five kernels (`L100`, `L100pc`, `L200`, `L200pc` and `L300`) in order to accommodate bigger problem sizes as discussed in following Subsection.

Listing 5.1: SDSoC OpenCL `L200-DF` implementation kernel code for the `L200` kernel.

```

.....
.....
__attribute__((xcl_dataflow))
//L200 kernel

//Data Read from DDR memory
//Read u,v and P
read_old(uold, vold, pold, local_uold, local_vold,
local_pold)
read_cu(cu, local_cu)
read_cv(cv, local_cv)
read_zh(z, h, local_z, local_h)

//Compute functions
//unew, vnew, Pnew
Compute:

local_unew (local_unew, local_uold, local_z, local_cv,
local_h);
local_vnew (local_vnew, local_vold, local_z, local_cu,
local_h);
local_pnew (local_pnew, local_pold, local_cu,
local_cv);

//Data Write back to DDR memory
//Write CU,CV, Z and H
write_unew(unew, local_unew)

```

```

write_vnew(vnew, local_new)
write_pnew(pnew, local_pnew)

```

```

.....
.....

```

Listing 5.2: SDSoC OpenCL L300-DF implementation kernel code for the L300 kernel

```

.....
.....
__attribute__((xcl_dataflow))
//L300 kernel

//Data Read from DDR memory
//Read u,v and P
read_uvp(u, v, p, local_u, local_v, local_p)
read_old(uold, vold, pold, local_uold, local_vold,
local_pold)
read_new(unew, vnew, pnew, local_unew, local_vnew,
local_pnew)

//Compute functions
//uold, vold, Pold
Compute:
local_uold (local_uold, local_unew);
local_vold (local_vold, local_vnew);
local_pold (local_pold, local_pnew);

//Data Write back to DDR memory
//Write uold, vold, Pold, u, v, P
write_old(uold, vold, pold, local_uold, local_vold,
local_pold)
write_old(u, v, p, local_unew, local_vnew, local_pnew)

.....

```


.....

5.1.2 Exploring the problem size

With having only five SWM kernels (L100, L100pc, L200, L200pc and L300) to implement, we have explored different problem sizes to find the highest possible one. As discussed in Section 5.1.1 we found that the largest possible problem size is (53*53) with the five kernels are optimised using the Data-Flow for L100, L200 and L300 kernels, and the pipeline mechanisms for the L100pc and L200pc kernels. This (53*53) problem size is used in the kernel-to-kernel exploration in both the SDSoC OpenCL and the Vivado approaches in Section 5.2.

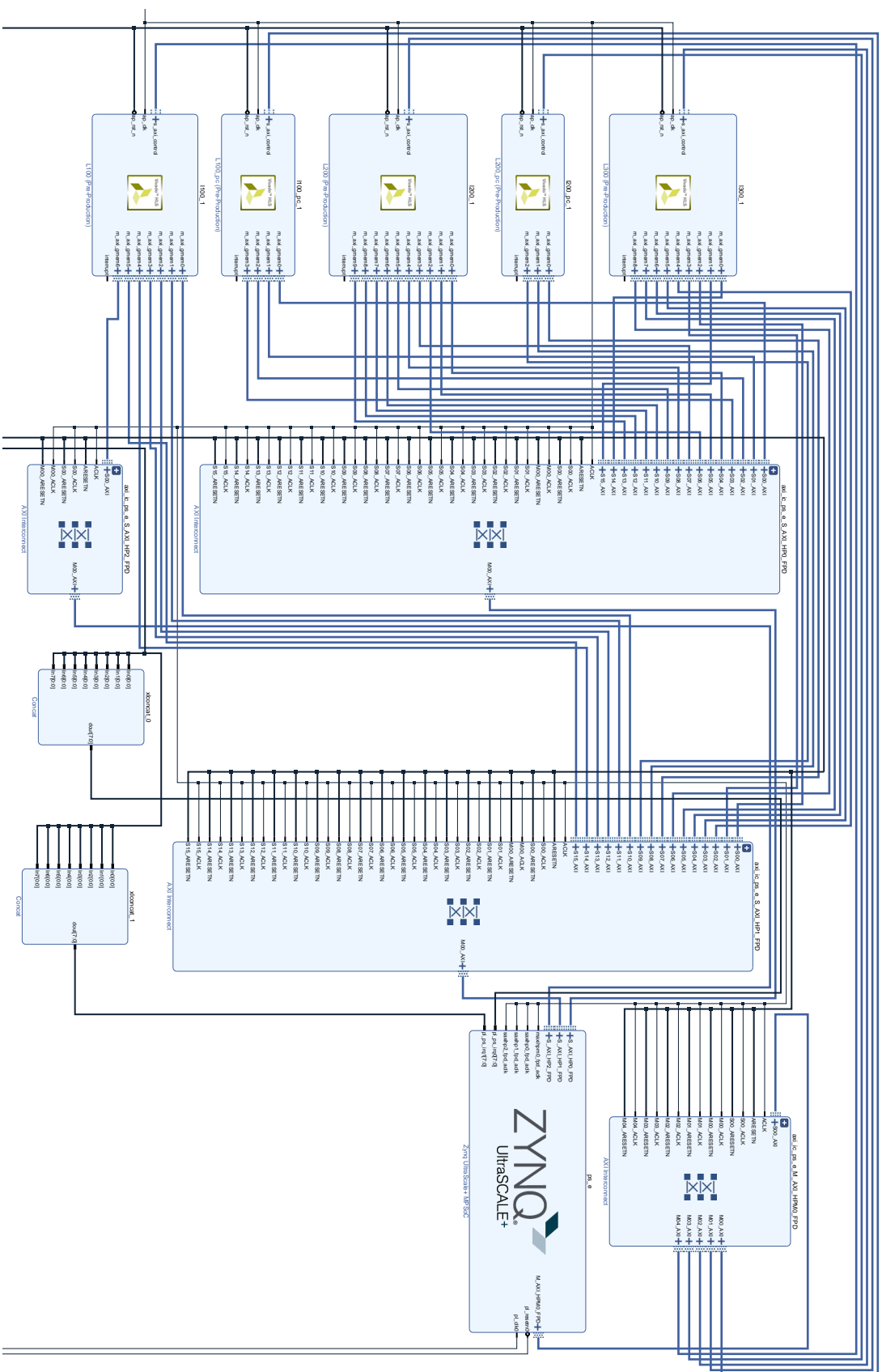
5.1.3 Kernel-to-kernel communication exploration

Figure 2.10 in Chapter 2 shows the data-movement behaviour between the five main kernels in the SWM application. There are two available mechanisms in the SDSoC OpenCL that can provide a design solution for the kernel-to-kernel data movements between the SWM five kernels. The two mechanisms are: the use of the DDR memory or the OpenCL pipes.

The use of the DDR memory solution means that the output data of a kernel is shared with the following kernel through the DDR memory. Each kernel starts with reading the input data from the DDR memory and writing the output data back to the DDR memory as input data for the next kernel. Figure 5.1 shows the system design for the SWM five kernels implementation with the use of the DDR memory mechanism for moving the data between the five SWM kernels.

The use of the OpenCL pipes solution means that the output data of a kernel is shared with the following kernel through the OpenCL pipes (AXI4_Stream Data FIFO). The first kernel e.g. (L100) starts with reading the input data from the DDR memory and then use the OpenCL pipe API function `write_pipe_block(<pipe_name>, array_name)` to write the output data to pipes. These pipes carries input data for the next kernel (L100pc). In the L100pc kernel, the OpenCL `read_pipe_block(<pipe_name>, array_name)` API function is used to read the input data. After the L100pc kernel's calculation phase finish, it writes the output data to another pipes that connected to the next kernel (L200), and so on for the other kernels; See code example in listing 5.3. As described in Sub-section 2.5.1 in the background Chapter 2, pipes have two modes, which are read/write blocking and unblocking modes. The mode used is blocking

Figure 5.1: The implementation system design for SDSoc OpenCL Five kernels with DDR mechanism for kernel-to-kernel communication implementation system design



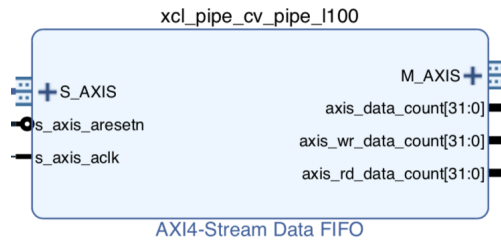


Figure 5.2: OpenCL Pipe IP Block example.

mode as it is the only mode supported by SDSoc Xilinx. The OpenCL pipe is created in the OpenCL kernel using the OpenCL attribute `pipe <data_type> <pipe_name> __attribute__((xcl_reqd_pipe_depth(size)))`.

Listing 5.3: A sketch of what the OpenCL code looks like when using pipes (just for two kernels interacting).

```

1 // OpenCL pipes creation
2
3 //L100 kernel's pipes
4 pipe float cu_pipe_l100
5     __attribute__((xcl_reqd_pipe_depth(16384)));
6 pipe float cv_pipe_l100
7     __attribute__((xcl_reqd_pipe_depth(16384)));
8 pipe float z_pipe_l100
9     __attribute__((xcl_reqd_pipe_depth(16384)));
10 pipe float h_pipe_l100
11     __attribute__((xcl_reqd_pipe_depth(16384)));
12
13 //L100pc kernel's pipes
14 pipe float cu_pipe_l100pc
15     __attribute__((xcl_reqd_pipe_depth(16384)));
16 pipe float cv_pipe_l100pc
17     __attribute__((xcl_reqd_pipe_depth(16384)));
18 pipe float z_pipe_l100pc
19     __attribute__((xcl_reqd_pipe_depth(16384)));
20 pipe float h_pipe_l100pc
21     __attribute__((xcl_reqd_pipe_depth(16384)));
22

```

```

23 //L100 kernel
24 __kernel void
25     __attribute__((reqd_work_group_size(1, 1, 1)))
26 l100(
27
28 //read u, v, p data from DDR memory
29 read_uvp(u,v,p);
30
31 //compute cu, cv, z, h
32 compute_cucvzh(cu, cv, z, h, u, v, p);
33
34 //pass cu, cv, z, h data to l100pc kernel
35 // through l100 kernel's pipes
36 __attribute__((xcl_pipeline_loop(1)))
37 for(int i=0; i< n_len*m_len; i++){
38 write_pipe_block(cu_pipe_l100, &cu[i]);
39 write_pipe_block(cv_pipe_l100, &cv[i]);
40 write_pipe_block(z_pipe_l100, &z[i]);
41 write_pipe_block(h_pipe_l100, &h[i]);
42 }
43 )
44
45 //L100pc kernel
46 __kernel void
47     __attribute__((reqd_work_group_size(1, 1, 1)))
48 l100(
49
50 //read cu, cv, z, h data from l100 kernel's pipes
51 // to local buffers
52 __attribute__((xcl_pipeline_loop(1)))
53 for(int i=0; i< n_len*m_len; i++){
54 read_pipe_block(cu_pipe_l100, &cu[i]);
55 read_pipe_block(cv_pipe_l100, &cv[i]);
56 read_pipe_block(z_pipe_l100, &z[i]);
57 read_pipe_block(h_pipe_l100, &h[i]);

```

```

58 }
59
60 //update cu, cv, z, h data boundary
61 update_cucvzh(cu, cv, z, h);
62
63 //pass the updated cu, cv, z, h data to the next kernel
64 // through l100pc kernel's pipes
65 __attribute__((xcl_pipeline_loop(1)))
67 for(int i=0; i< n_len*m_len; i++){
68 write_pipe_block(cu_pipe_l100pc, &cu[i]);
69 write_pipe_block(cv_pipe_l100pc, &cv[i]);
70 write_pipe_block(z_pipe_l100pc, &z[i]);
71 write_pipe_block(h_pipe_l100pc, &h[i]);
72 }
73)

```

Table 5.1: The compute time detail for the three SDSoC OpenCL SWM 5 Kernels optimized versions implementations Versus the SDSoC OpenCL SWM 5 Kernels un-optimized version. All versions implemented on ZCU102 FPGA board with a clock frequency of 200 MHz (The maximum possible frequency). In the "DDR" implementation the host will explicitly start kernels when previous kernels have completed, so there is no kernel to kernel marshalling here. Un-optimised SWM kernels implementations are similar to the L100_seq implementation in Chapter4.

Kernels Timing (seconds)	SWM- 5Kernels- (53*53) un-optimised	SWM- 5Kernels- (53*53) DDR	SWM- 5Kernels- (53*53) PipesV1	SWM- 5Kernels- (53*53) PipesV2
L100	124.961	0.829	0.635	0.834
L100pc	4.511	0.334	0.431	0.475
L200	95.152	0.671	0.702	0.732
L200pc	3.655	0.320	0.457	0.464
L300	100.235	0.781	0.783	1.774
Total kernels Only Compute Time	328.514	2.935	3.008	4.279

Two SWM five kernels OpenCL pipes versions are explored in the following. The first version is named SWM-5Kernels-PipesV1 and the second is SWM-5Kernels-PipesV2 in Table 5.1. Figure 5.3 shows the system design for the first pipe (SWM-5Kernels-PipesV1)

design version. In this first pipe implementation, we have created fourteen pipes. Four between the L100 and L100pc kernels, four between the L100pc and L200 kernels, three between the L200 and L200pc kernels, and three between the L200pc and L300 kernels, see Figure 5.3. These kernels are executed 4000 times, so in each iteration, the execution of the kernels starts with L100 kernel reading input data from the DDR memory, then the output data is shared between the following kernels through the pipes. At the end of an iteration, kernel L300 writes the output data back to the DDR memory to act as input data for the L100 kernel in the next iteration. This design option requires eighteen DDR memory accesses² (12 read access, 6 write access) and fourteen pipe accesses in every iteration of the 4000 iterations. L100 kernel does three DDR memory read accesses to read *u*, *v* and *P* data; L200 kernel does three DDR memory read accesses to read *uold*, *vold* and *Pold* data; and L300 kernel does six DDR memory read accesses to read *u*, *v*, *P*, *uold*, *vold* and *Pold* data. The six DDR memory write accesses are happening at the end of the L300 kernel to write the new *u*, *v*, *P*, *uold*, *vold* and *Pold* data for the next iteration.

In the second pipe design version (SWM-5Kernels-PipesV2), we explored the option to minimize the access to the DDR memory and flow the data between the five kernels through only pipes when possible. Figure 5.4 shows the system design for the second pipe design option, where we have created twenty pipes. Compared to the first pipe design option, we have introduced six new pipes. Figure 5.4 shows that three pipes connect the L300 kernel with the L100 kernel, and three connect L300 kernel with the L200 kernel. These six pipes supply the new *u*, *v* and *P* data for the L100 kernel; and *uold*, *vold* and *Pold* data for the L200 kernel every iteration. L300 kernel is a key factor for this design, as it produce the new output for the next iteration. As described in Section 2.6.1 from Chapter 2, the execution of L300 kernel starts in the second iteration. The SWM application starts in the first iteration by executing the L100, L100pc, L200 and L200pc kernels, then a host side execution of L300pc kernel before the start of the second iteration. In the first iteration (in both pipes designs), the L200pc output data need to be written back to the host for the second iteration initialisation. Therefore, to adapt this second pipe design option with the dataflow behaviour of the shallow application, we have used flags within each kernel code to control the data access either to/from DDR memory or pipes (depending on the iteration number).

L300 kernel reads and writes the *u*, *v*, *P*, *uold*, *vold* and *Pold* data. In this second

²Each of these "accesses" is a variable read/write and comprises very many individual memory accesses (each 32-bits wide).

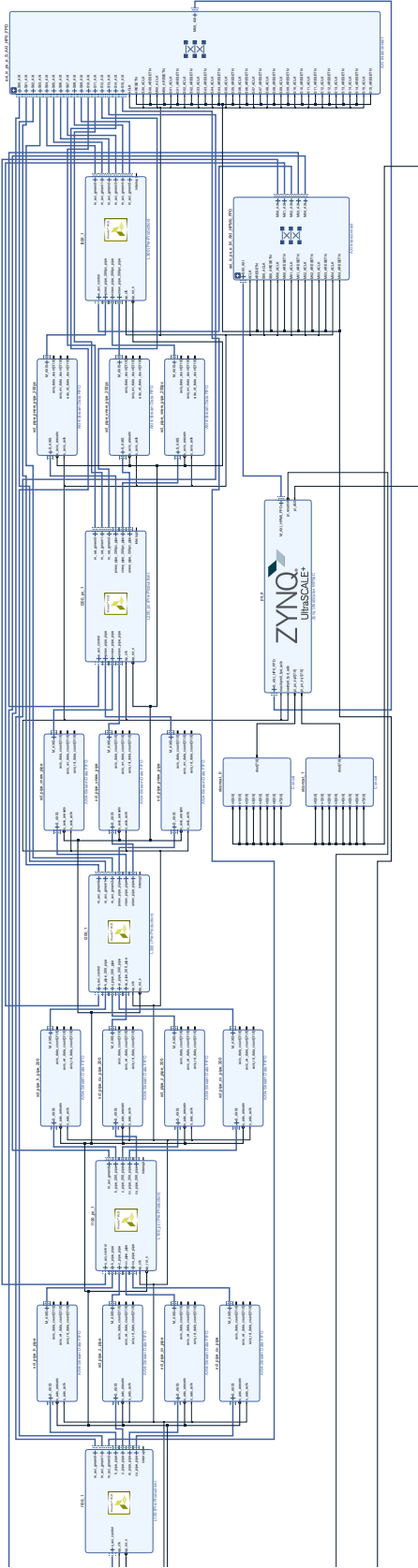


Figure 5.3: The implementation system design for the SDSoC OpenCL Five kernels with Pipes (Version 1) mechanism for kernel-to-kernel communication implementation system design

pipe version these data are shared with the L100 and L200 kernels through pipes to keep data flow between the kernels. L300 kernel also needs a copy of these data as its input data for the next iteration. L300 kernel cannot pass data to itself for the next iteration through pipes because we are using the dataflow design and parallel read/write to a pipe at the same time violates the dataflow design. Therefore, in this case we store a copy of the `u`, `v`, `P`, `uold`, `vold` and `Pold` data in the DDR memory to be used as input data for the L300 in the next iteration. This is the only place where we access the DDR memory. From iteration number two, the data is streaming between the five kernels only through the twenty created pipes, since it was anticipated that pipes would provide a more efficient mechanism of sharing data than the use of DDR memory. This design option requires twelve DDR memory accesses (six read access, six write access), compared to eighteen in the first pipe option, and twenty pipes accesses in every iteration (from iteration two) of the 4000 iterations.

Table 5.1 shows the compute time of the three explored kernel-to-kernel SDSoC OpenCL optimised implementations. In addition, the table shows a initial SWM five kernels un-optimized implementation (SWM-5Kernels-un-optimised) that we compare against. The focus in this section is on the effect on the kernel's compute time of the kernel-to-kernel mechanism options, and the use of the dataflow method, as kernel optimization techniques; however, in Section 6.2 from Chapter 6 we discuss the performance from the perspective of the overall application time of these three implementations, including the data movement overhead time.

The use of DDR memory (SWM-5Kernels-DDR) as the data-sharing mechanism between the five kernels achieved a compute time of 2.934s, which is faster by 111.92x compared to the (SWM-5Kernels-un-optimised) un-optimised version (328.514), see Table 5.1. The use of DDR memory as the kernel-to-kernel mechanism also provided a slightly better compute time (by 1.02x) than the (SWM-5Kernels-PipesV1) implementation (3.008s) and 1.45x faster than the (SWM-5Kernels-PipesV1) implementation (4.279s).

The compute time figures in Table 5.1 show a performance difference between the use of DDR memory and the pipes. The bandwidth access between these two technologies is different. DDR memory provides a higher bandwidth by utilizing four memory ports, while pipes offer only one producer and one consumer port. In addition, the blocking pipe mode (the only supported mode) limited the overlapping between the execution of the kernels. The execution time of the two pipes implementations was limited to the pipes' read/write access rate.

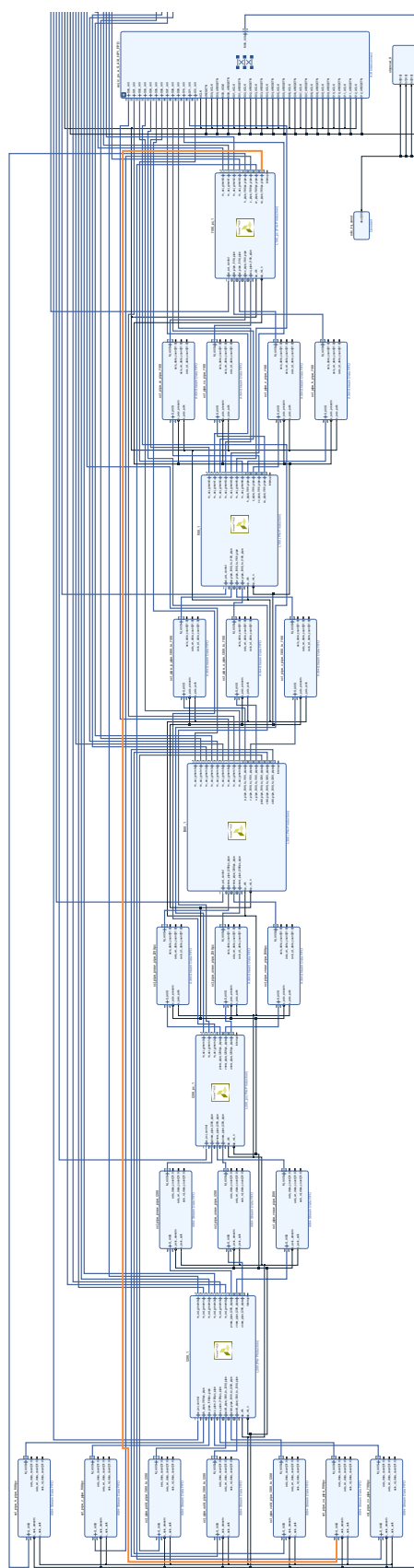


Figure 5.4: The implementation system design for the SDSoC OpenCL Five kernels with Pipes (Version 2) mechanism for kernel-to-kernel communication implementation system design

In comparing the two pipes kernel-to-kernel implementations, Table 5.1 shows that the pipe version1 compute time was faster (3.008s) than the pipe version2 implementation (4.279s) by 1.42x. It can be seen from Table 5.1 that the biggest source of overhead in the second pipe design option is the L300 kernel. As described earlier, L300 performs six DDR memory write accesses and six pipes write access operations. In addition, it does three pipes read accesses and six DDR memory read access operations, see Figure 5.4. This mixed-use of DDR and pipes in the L300 kernel contributes to slowing down the kernel's execution time.

Table 5.2: Resource Usage Figures of the SDSoC OpenCL SWM Five Kernels implementations.

Experiments Name	FF	LUT	DSP	BRAM_36K
SWM-5Kernels un-optimized	30657 (5.59 %)	38266 (13.96%)	52 (2.06%)	14 (1.53%)
SWM-5Kernels DDR	107159 (19.54%)	127365 (46.47%)	262 (10%)	462 (50.65%)
SWM-5Kernels Pipes-V1	123315 (22.49%)	118055 (43.07%)	261 (10%)	408 (44.73%)
SWM-5Kernels Pipes-V2	81250 (14.82%)	91900 (33.53%)	199 (8%)	406 (44.51%)

In terms of resource usage, Table 5.2 shows that the the DDR implementation required slightly more BRAM and LUT resources compared to the pipe implementations. The pipes version1 implementation FF, LUT and DSP resources utilisation were more than the resources utilised in the pipes version2 implementation. Both pipe implementations required a similar amount of BRAM blocks. The DDR memory mechanism design was more straightforward in terms of programmability as it required no extra coding and effort compared to the pipes versions. The use of pipes required high coding effort and raised many debugging issues related to the halting of the application's execution due to synchronisation issues.

5.2 SWM Application Mapping Using Vivado

5.2.1 Optimise the SWM Vivado kernels

The mapping exploration of the L100 kernel concurrency using the Vivado approach in Section 4.4 from Chapter 4 shows that the use of Data-Flow mechanism provided

the best performing implementation; however, this implementation required the use of multiple external BRAM blocks. Applying the Data-Flow mechanism on the L200 and L300 kernels, as we did in the SDSoC OpenCL multiple kernels in Section 5.1, would exhaust the BRAM resources available and increase the complexity of the Vivado host code design and the system design. The management of the external BRAM addresses in the Vivado approach has to be conducted manually (this is done automatically using the OpenCL approach), therefore, using Data-Flow would require a great deal of programmer effort in the Vivado multiple kernel implementation. Instead, we decided to apply the pipeline mechanism as we did in the L100-P-a implementation (described in Sub-section 4.4.2 from Chapter 4) with the use of code option (a) from Figure 4.1 in Section 4.1 from Chapter 4 on all the five SWM kernels.

5.2.2 Finding the problem size

Similar to what we determined in Section 5.1, the largest problem size we found with the five kernels are optimized using the pipeline mechanisms is (53*53), which is, therefore, is the used size in the following sub-section.

5.2.3 Kernel-to-Kernel Communication Exploration

There is only one implementation explored in this Section and this is motivated by the lessons learned from the Vivado exploration study in Section 4.4 from Chapter 4. In addition, we concentrate on shared memory solutions between the HLS kernels rather than HLS streams based on lessons learned from the OpenCL implementation in section 5.1.3, which proved to be the most efficient kernel design. Figure 5.5 shows the system design of the Vivado five kernels implementation. The five kernels are optimised using the pipeline option, and each kernel is compiled in isolation, resulting in five IP blocks. All the five IP blocks share only a single external BRAM block in this design. We have learned from the exploration study in Section 4.4 from Chapter 4 that the use of multiple external BRAM blocks with multiple kernels can provide a slight performance improvement, as was seen with the L100-P-a implementation compared with the L100-M-P-4-BRAMs-Blocks in Table 4.8 which discussed the cost of duplicating array data. We thus decided only to use one external BRAM block in this implementation to avoid increasing the design and programming complexity as well as the associated increase in resources cost for only a slight performance benefit.

Table 5.3 shows that this implementation's execution time is 1.340 seconds, and it

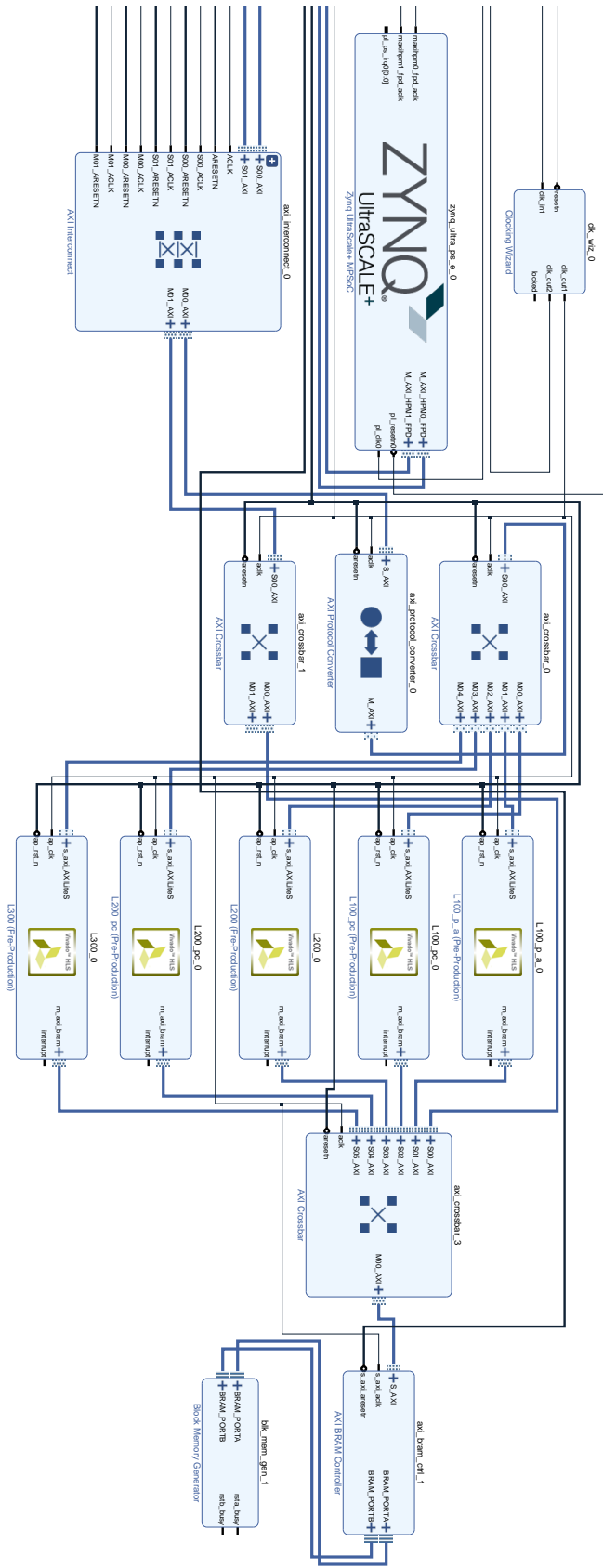


Figure 5.5: The implementation system design for the Vivado Five kernels with one external BRAM method for kernel-to-kernel communication

consume 71% of the BRAM resources, as reported in Table 5.4. The BRAM resources of this implementation is consistent with our choice of applying only the pipeline mechanism and using a single external BRAM block. The use of Data-Flow or multiple external BRAMs would likely have exhausted the BRAM available and thus caused a compiler error.

Table 5.3: Performance Figures of the Vivado SWM 5 kernels implementation.

Kernels Timing (Seconds)	SWM-5Kernels- P-a
Freq MhZ	447
L100	0.234
L100pc	0.214
L200	0.308
L200pc	0.163
L300	0.415
Total Execution Time	1.340

Table 5.4: Resource Usage Figures of the Vivado SWM Five Kernels implementation.

Experiments Name	FF	LUT	DSP	Total Local & External BRAM 36K Usage
SWM-5Kernels- P-a	48988 (8.93%)	36425 (13.28%)	112 (4%)	247 (71%)

5.3 Summary

This chapter explored the mechanisms available in the SDSoc OpenCL, and Vivado approaches to map the five SWM kernels to the ZCU102 FPGA board. We also explored the design options and mechanisms in the two approaches for managing the data movement between the five SWM kernels. The trade-offs in the different approaches, in terms of performance and resource usage, were discussed, along with issues related

to programmability. The results of this exploration study showed that in the multiple kernels case, utilising the `Data-Flow` mechanism efficiently depends on the memory design solution. We found that using DDR memory in the OpenCL design helped provide enough memory bandwidth with multiple `Data-Flow` kernels compared to the Vivado multiple `Data-Flow` kernels solution. The use of an external BRAM solution in the Vivado design was not suitable for `Data-Flow`-style kernels, which we trade with `pipeline` kernels style. The results also revealed that the use of the shared-memory method (DDR and external BRAM) between multiple kernels provided better performance against the another explored data movement methods (pipes). This design decision was the best in both the SDSoc OpenCL and the Vivado kernel-to-kernel communication exploration studies. In addition, it was the most straightforward design solution in terms of programmability.

Chapter 6

Comparison Study (1): SWM Implementations In SDSoc OpenCL Versus Vivado

This chapter compares the data gathered from exploration study (1) described in Chapter 4 and Chapter 5 for mapping the concurrency types in the SWM application to the ZU9102 FPGA board using the SDSoc OpenCL and the Vivado approaches. Here, we compare the data from exploration study (1) related to two main aspects, with the use of the quantitative and qualitative metrics discussed in Section 3.4 in Chapter 3. The first aspect (in Section 6.1) considers the data from the *L100 mapping explorations*. The second aspect (in Section 6.2) considers the data from the *kernel-to-kernel communication exploration* in the multiple kernel implementations. We also discuss the differences arising from the use of the two approaches (SDSoc OpenCL and Vivado) and their effect on the achieved performance. These differences arise from two primary sources: first, from the different processes required to create and implement the designs and the implied use of FPGA resources; and, secondly, from timing constraints in the designs which dictate the highest frequency at which a design can execute.

6.1 L100 concurrency mapping comparison

This section compares the L100 concurrency mapping performance and resources usage results from Chapter 4 using the comparison metrics described in Subsection 3.4.3 from Chapter 3. The comparison is conducted between the best mapping mechanisms from both the SDSoc OpenCL and the Vivado approaches.

6.1.1 Performance Analysis

Table 6.1 presents the application runtime details for the best L100 SDSoC OpenCL and Vivado implementations from the exploration study (1) in Chapter 4. In Chapter 4 we presented the **compute time** (kernel computational time) in Seconds of the mapping implementations. In this section, we are comparing the **compute time** between the implementations from the two HLS approaches and the application **overall time** in Seconds that also includes the data movement time. The **Overall time** metric is presented in this chapter in the context of each the three main phases of the application runtime: Load Time, preparing and loading data from the host to the FPGA memory solution, Compute Time, computing, and Store Time, returning the data back from the FPGA memory solution to the host. We also present the compute time in the form of flops/s for only the best SDSoC OpenCL and Vivado implementations. The methods used to calculate the two sets of figures in Table 6.1 are different and are discussed further in this Section.

Table 6.1: The application runtime details of the best L100 SDSoC OpenCL and Vivado implementations in Chapter 4.

HLS Approach	Implementation Name	Frequency MHz	Load Time(s)	Compute Time(s)	Store Time(s)	Overall Time(s)
L100 SDSoC OpenCL	L100-P-a	200	0.391	0.661	2.312	3.364
	L100-DF	200	0.387	0.601	2.304	3.292
	L100-M-P	200	0.408	1.716	2.330	4.454
L100 Vivado	L100-P-a*	200	0.706	0.350	11.313	12.369
	L100-P-a	447	0.635	0.264	10.369	11.868
	L100-DF-7-BRAMs	447	4.061	0.164	11.138	15.363
	L100-M-P 4-BRAMs	447	2.312	0.238	11.308	13.858

Compute Time Metric: In comparing the computational time, Table 6.1 show that the best L100 mapping implementations provided the best compute time in Chapter 4 is the Vivado L100-DF-7-BRAMs. This design achieved a compute time of 0.164 seconds. Table 6.1 also shows that the Vivado L100 implementations achieved better compute time compared to the SDSoC OpenCL implementations. Two factors played a major role in the achieved Vivado L100 compute time. The first factor is that the Vivado implementations are executed with a clock frequency of 447 MHz compared to only

200 MHz on the SDSoC OpenCL L100 implementations. Nonetheless, executing a Vivado implementation with a 200 MHz frequency for comparison (See implementation L100-P-a* in Table 6.1), shows that the Vivado L100 implementation still provides a better compute time, which leads to the second factor. The second factor that provided the Vivado implementation with better compute time is that the Vivado L100 IP block accesses data from an external BRAM block, rather than from DDR memory access, as in the SDSoC OpenCL L100 design. Both SDSoC OpenCL and Vivado L100 kernel designs in terms of the levels of optimisations and mapping techniques are actually identical and read/write the same amount of input/output data. However, the L100 IP block computational time is affected by the external BRAM's data access time versus DDR memory access time. The compute time in the SDSoC OpenCL L100 design is slower because it includes the overhead involved in accessing data from the slower DDR memory option.

Computation Flop Rate (flop/s): Analysis of the L100 design timeline reveals that in the compute part of the best L100 Vivado (L100-DF-7-BRAMs) design; there are a maximum of seven flops per cycle performed. At 447 MHz, this leads to a theoretical peak performance figure of 3.1 Gflop/s. However, based on the kernel achieved compute time, see Table 6.1, the Vivado (L100-DF-7-BRAMs) design achieved 1.4 Gflop/s¹(45.16% of peak)². In a similar analysis for the timeline of the best performing L100 SDSoC OpenCL design (L100-DF), there are a maximum of six flops per cycle performed. Meaning, that at 200 MHz, this leads to a theoretical peak performance figure of 1.2 Gflop/s. However, based on the compute time figure the L100-DF design, see Table 6.1, has only achieved 655.36 Mflop/s of performance (54.5% of peak).

Overall Time Metric: This metric is used to compare the overall application execution time which includes the load time, plus the compute time and store time of the best L100 SDSoC OpenCL and Vivado implementations presented in Table 6.1. In both the SDSoC OpenCL and the Vivado L100 implementation designs, we load and store the same amount of data, except in some specific Vivado implementations. However, the data movement methods used are not equivalent due to the different memory solutions used.

The data movement method between the SDSoC OpenCL host and the OpenCL

¹The actual achieved flops per second performance is calculated using this simple calculation: total-flops (24flops*problem size(64*64)*number of iterations(4000)/ Actual run time)

²Theoretical peak performance is calculated using this simple calculation: Frequency * the number of flops per cycle. Applying this to the best L100 Vivado L100-DF-7-BRAMs design (447*7 = 3.1 GF/s)

L100 FPGA IP block is managed as follows. The OpenCL system creates two DDR memory spaces where one space is allocated for the host code buffers, and the other space is allocated for the OpenCL kernel's buffers (known as global memory in the OpenCL terminology). The OpenCL API `EnqueueRead/WriteBuffer` functions are used in the SDSoC OpenCL L100 host code to first write the input data to the OpenCL buffers (which measured in in the code as `Load Time`) and when the kernel finished execution (measured as `Compute time`), we read the output data back to the host buffers from the OpenCL buffers (measured as `Store Time`), as can be seen in code Listing 6.1. These three phases represent the SDSoC OpenCL L100 overall time, which are executed for 4000 time-steps. The different SDSoC OpenCL L100 implementations are Loading and Storing the same amount of data and use the same data movement method. Table 6.1 shows that the `Load` and `Store` time of the SDSoC OpenCL implementations are similar.

Listing 6.1: L100 SDSoC OpenCL Load and Store API functions.

```

.....
.....

//main loop of 4000 timesteps
for ( ncycle=1;ncycle <=4000;ncycle++) {

//Write L100 kernel input data to DDR
//record load time
load=clock();
q.enqueueWriteBuffer(buffer_p , CL_TRUE, 0,
vector_size_bytes , source_p.data());
q.enqueueWriteBuffer(buffer_u , CL_TRUE, 0,
vector_size_bytes , source_u.data());
q.enqueueWriteBuffer(buffer_v , CL_TRUE, 0,
vector_size_bytes , source_v.data());
q.finish();
//accumulate load time
load_dur += ( clock() - load ) / (double) CLOCKS_PER_SEC;

//L100 Kernel launch

```

```

//record compute time
l100_s = clock();
kernel_l100(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1),
cl::NDRange(1, 1, 1)), buffer_u, buffer_v,
buffer_p, fsdx, fsdy);
q.finish();
//accumulate compute time
l100_dur += (clock() - l100_s) / (double) CLOCKS_PER_SEC;

//Read output data from DDR
//record store time
store=clock();
q.enqueueReadBuffer(buffer_cu, CL_TRUE, 0,
vector_size_bytes, source_cu.data());
q.enqueueReadBuffer(buffer_cv, CL_TRUE, 0,
vector_size_bytes, source_cv.data());
q.enqueueReadBuffer(buffer_z, CL_TRUE, 0,
vector_size_bytes, source_z.data());
q.enqueueReadBuffer(buffer_h, CL_TRUE, 0,
vector_size_bytes, source_h.data());
q.finish();
//accumulate store time
store_dur += (clock() - store) / (double) CLOCKS_PER_SEC;

.....
.....
} //end of 4000 timesteps

```

In contrast, the data movement method between the Vivado host and the Vivado L100 FPGA IP block is managed in the L100 Vivado designs as follows. The host prepares and transfers the input data to the external BRAM block in the Vivado system design using manually-managed memory address configuration. As shown in Figure 4.7 in Chapter 4, the ZynQ IP block has a direct connection to the external BRAM block. This helps to have a separate communication path between the host code and

the external BRAM block. The Vivado host code fills first the external BRAM block with the input data using the `load_input` function in code listing 6.2 (Load Time). Following that, the kernel starts execution by fetching the input data from the external BRAM block to local BRAM buffers and computes the kernel's operations (Compute time). In the final phase, the host code reads back the output data from the external BRAM block to the host buffers using the `get_output` function in code listing 6.2 (Store Time). These three phases, which are executed for 4000 timesteps, contribute to the Vivado L100 overall time. The different Vivado L100 implementations are Loading and Storing data with the same data movement method. Table 6.1 shows that the Vivado implementation's Store time are similar. However, the Load time is different depending on the number of created external BRAM blocks. The Vivado designs with one external BRAM block have a similar Load time. However, designs such as L100-DF-7-BRAMs and L100-M-P-4-BRAMs have different amounts of input data as the design decision included duplicating and distributing the input data in different external BRAM blocks. This decision resulted in increasing the Load Time cost.

Listing 6.2: L100 Vivado Load and Store functions.

```

.....
.....

//main loop of 4000 timesteps
for ( ncycle=1;ncycle <=4000;ncycle++) {

//Write L100 kernel input data to external BRAM
//record load time
load=clock();
int load_input (int ib, float *u,
float *v, float *p){
for (i=0;i<N_LEN*M_LEN;i++) {
*(sfpga_u[ib]+i) = u[i];
*(sfpga_v[ib]+i) = v[i];
*(sfpga_p[ib]+i) = p[i];
}
//accumulate load time
load_dur += ( clock() - load ) / (double) CLOCKS_PER_SEC;

```

```

//L100 Kernel launch
//record compute time
l100_s = clock();
ierr = fpga_start (0);
ierr= fpga_wait(0);
//accumulate compute time
l100_dur += (clock() - l100_s) / (double) CLOCKS_PER_SEC;

//Read output data from external BRAM
//record store time
store=clock();
int get_output(int ib, float *cu,
float *cv, float *z, float *h ) {
for (i=0;i<N_LEN*M_LEN;i++) {
*(cu+i) = *(sfpga_cu[ib]+i);
*(cv+i) = *(sfpga_cv[ib]+i);
*(z+i) = *(sfpga_z[ib]+i);
*(h+i) = *(sfpga_h[ib]+i);
}
//accumulate store time
store_dur += (clock() - store) / (double) CLOCKS_PER_SEC;

.....
.....
} //end of 4000 timesteps

```

The overall times in Table 6.1 show that all three SDSoC OpenCL L100 designs provided faster overall application times compared to that of the Vivado designs. There is a significant overall time difference between the SDSoC OpenCL and the Vivado designs because the Vivado implementations's Load time and Store time are slower than the SDSoC OpenCL implementations's Load time and Store time. Table 6.1 shows that the Vivado design that utilised one external BRAM block such as L100-P-a has a Load time that is slower by 1.64x and a Store time that is slower by nearly

4.5x compared to the fastest overall SDSoC design (L100-DF); while implementations with multiple external BRAM blocks such as L100-DF-7-BRAMs and L100-M-P-4-BRAMs have a Load time that is slower by 10.49x and 5.97x respectively, and Store time that is slower by nearly 4.8x for both implementations. The Load and Store time differences between the designs from both approaches is a result of the different memory path between the host and the memory solutions utilized for the L100 kernel(i.e. DDR versus external BRAM). The data movement paths between the SDSoC OpenCL host DDR memory space and the SDSoC OpenCL L100 kernel's DDR memory space is shorter compared to the data path between the Vivado host DDR memory and the external BRAM. The SDSoC OpenCL design transfers the data within the same memory between two DDR memory spaces (host buffers and OpenCL buffers). In contrast, the data in the Vivado designs travel a longer distance between the DDR memory (host buffers) and the external BRAM block (kernel buffers) inside the FPGA fabric area. In conclusion when considering the analysis of the overall time metric, although the SDSoC OpenCL implementations in Table 6.1 did not provide the best compute time, the SDSoC OpenCL L100-DF implementation provided the best overall time of 2.292s compared to the other SDSoC L100 implementations and to the Vivado L100 implementations.

6.1.2 Resource Usage Analysis

Table 6.2 shows the resource usage for the best SDSoC OpenCL and Vivado L100 implementations from the exploration study (1) in Chapter 4. The figures show that the most utilised resources are BRAM blocks. The Data-Flow implementations in both the SDSoC OpenCL and the Vivado approaches utilised the highest BRAM resources (27%) and (31%), respectively, due to the design decision taken in the Data-Flow implementations.

6.2 Multiple kernel mapping comparison

This section compares results from the SDSoC OpenCL and the Vivado SWM five Kernel implementations from the SWM multiple-kernels mapping exploration conducted in Chapter 5. We compare the performance (Compute time and Overall time), the kernel-to-kernel data movement methods, and the resource usage figures of the three best implementations from Chapter 5, which are summarised in Table 6.3.

Table 6.2: Total design resource usage figures for the best L100 SDSoC OpenCL and Vivado implementations in Chapter 4.

HLS Approach	Implementation Name	FF	LUTs	DSPs	BRAM_36K
L100 SDSoC OpenCL	L100-P-a	23593 (4%)	20653 (8%)	55 (2%)	84 (9%)
	L100-DF	28086 (5%)	27819 (10%)	102 (4%)	248 (27%)
	L100-M-P	41948 (8%)	38278 (14%)	89 (3%)	171.50 (19%)
L100 Vivado	L100-P-a*	14697 (3%)	11768 (4%)	55 (2%)	78.50 (8.61%)
	L100-P-a	22267 (4%)	16180 (6%)	39 (1%)	83 (9%)
	L100-DF-7-BRAMs	30238 (6%)	22285 (8%)	65 (2%)	283.50 (31.09)
	L100-M-P 4-BRAMs	29712 (5%)	20052 (7%)	65 (2%)	195.50 (21%)

Table 6.3: The application runtime details for the SWM five Kernels implementations in SDSoC OpenCL and Vivado approaches in Chapter 5.

HLS Approach	Implementation Name	Frequency MHz	Load Time(s)	Compute Time(s)	Store Time(s)	Overall Application Time(s)
SDSoC OpenCL	SWM-5Kernels-(53*53) DDR	200	0.000609	2.935	0.000704	2.936
	SWM-5Kernels-(53*53) PipesV1	200	0.000609	3.008	0.000704	3.0093
Vivado	SWM-5Kernels-P-a	447	0.000472	1.334	0.000284	1.337

6.2.1 Performance Analysis

Compute Time Metric: As was discussed in Chapter 5, the SWM five kernels designs in both approaches are different in terms of utilized optimization levels and mapping

mechanisms. The mapping mechanism used in the Vivado SWM five kernels implementation was the pipeline mechanism, while the Data-Flow mechanism was used to map the five SWM kernels in the SDSoC OpenCL implementations. In comparing the computational time, Table 6.3 shows that the compute time of the Vivado SWM five kernels implementation provided the best compute time (1.334s) compared to the two SDSoC OpenCL SWM five kernel implementations (2.935s (DDR) and 3.008s (pipes)).

For a similar reason to that discussed in Section 6.1, the Vivado implementation has higher performance because it runs at a higher clock frequency (447 MHz) compared to only 200 MHz used in the SDSoC OpenCL implementations. In addition, the Vivado implementation accesses a faster memory solution, using an external BRAM block rather than the DDR memory used in the SDSoC OpenCL designs.

Overall Time Metric: In both the SDSoC OpenCL and the Vivado SWM five kernels implementation designs (DDR and pipes), we `Load` and `Store` the same amount of data. In addition, the `Load` and the `Store` operations are executed only once at the start of the 4000 time-steps. In both SDSoC OpenCL and the Vivado implementations we `Load` the data before the 4000 time-steps start, then the compute phase for the five kernels is executed for 4000 time-steps and at the end of the compute phase we `Store` the results back from the FPGA kernel. The output data from each kernel are shared between the kernels over time-steps through the utilised kernel-to-kernel memory solution. Three `kernel-to-kernel` design options were explored in Chapter 5 for the OpenCL SDSoC and Vivado SWM five kernels implementations. These design solutions utilised DDR memory and pipes in the SDSoC OpenCL implementations and an external BRAM block in the Vivado design. In both cases, Kernels read and write through a form of shared memory solution to avoid having to share the data through the host code. Table 6.3 shows that the time cost of the `Load` and `Store` phases is, in all cases, small compared to the compute time. In addition, Vivado `Load` time and the `Store` time are faster by around 1.29x and 2.47x respectively compared to the two SDSoC OpenCL implementation's `Load` and `Store` time.

In conclusion to the overall time metric analysis, Table 6.3 shows that the Vivado SWM five kernels implementation with the pipeline mechanism and one shared external BRAM block has provided the best overall application time between the explored implementations in Chapter 5. This Vivado implementation was faster by 2.2x and 3.2x than the SDSoC OpenCL DDR and SDSoC OpenCL pipe versions, respectively.

6.2.2 Resource Usage Analysis

Table 6.4 shows the resource usage figures for the best SWM five kernels implementations from both the SDSoC OpenCL and the Vivado approaches. The Vivado SWM five Kernels implementation consumed the lowest number of FF, LUTs and DSPs resources, compared to the SDSoC OpenCL SWM five Kernels with DDR implementation, and the pipes-V2 implementation. The SDSoC OpenCL SWM five Kernels pipe version consumed 78% of the BRAM resources (the highest) followed by a consumption of 71% of the BRAM resources in the Vivado design. The comparison results in this section show that the Vivado SWM five kernels implementation not only achieved the best performance but required the lowest resource usage.

Table 6.4: Total design resource usage figures for the best SWM five Kernels implementations in SDSoC OpenCL and Vivado approaches in Chapter 5.

HLS Approach	Implementation Name	FF	LUTs	DSPs	BRAM_36K
SDSoC OpenCL	SWM -5kernels -DDR	162665 (30%)	155261 (57%)	262 (10%)	504 (55%)
	SWM -5kernels -Pipes-V2	1180011 (22%)	123529 (45%)	199 (8%)	715 (78%)
Vivado	SWM -5Kernels -P-a	56309 (10%)	37189 (14%/)	112 (4%)	649 (71%)

6.3 Development Effort and Hardware Level of Expertise

The exploration study (1) reported in Chapter 4 and in Chapter 5 showed that the L100 mapping mechanism that required the highest amount of development effort in both the SDSoC OpenCL and the Vivado approaches was the Data-Flow mechanism. Although this mechanism provided the best compute time and overall time, it required significant coding changes to the initial implementation and coding effort (to avoid introducing coding bugs). However, the pipeline mapping mechanism with the single-nested-loop-based coding option was the most straightforward choice to implement for the Vivado SWM multi-kernels implementation.

In terms of kernel-to-kernel communication development effort described in Chapter 5, the use of the DDR memory option required the lowest development effort as it required less host and kernel coding effort and was straightforward to implement. Utilising pipes as a kernel-to-kernel communication solution required a great deal of coding effort in both the SDSoC OpenCL host and kernel codes.

6.4 Summary

This chapter compared the data gathered from the exploration study (1) conducted in Chapter 4 and Chapter 5. The comparison analysed the performance results, resource usage figures, and design decisions for the L100 kernel concurrency mapping study and the SWM five kernels exploration in both the SDSoC OpenCL and the Vivado approaches. The comparison results revealed the following points. The Data-Flow mechanism with functions provided the best performance for mapping the L100 kernel's concurrency in both SDSoC OpenCL and Vivado approaches. However, this mechanism required high BRAM resource usage and coding effort.

The L100 Data-Flow Vivado design achieved the best compute time and the L100 SDSoC OpenCL Data-Flow design provided the best overall application time (3.292s). The data movement Load and Store cost was high compared to the compute time in the both SDSoC OpenCL and Vivado L100 implementations, and this dominated the overall time. However, the use of the DDR memory in the SDSoC OpenCL implementations provided better data movement (Load, Store) times compared to the Vivado L100 (Load, Store) multi times. Nevertheless, in the SWM five Kernels implementations, the data movement cost was less significant, not affecting either approach's design compute time, because the data were only shared between the five kernels through three data-movement solutions options (DDR, Pipes, and external BRAM). The external BRAM memory block shared between the five kernels in the Vivado design provided the best compute and overall time in the kernel-to-kernel communication comparison.

In summary, comparing the SDSoC OpenCL and the Vivado design methodologies, the Vivado design methodology with external BRAM solutions required higher development effort and hardware expertise. It requires the programmer to take care of low-level concerns to create the FPGA solution, including the manual creation of the FPGA hardware system design, which requires low-level configuration, connections between the IP blocks and address manipulation management. In managing the host

and the kernel solutions, the Vivado approach required more coding effort and FPGA expertise.

Chapter 7

Exploration Study (2): *MatVec* Kernel with SDSoC OpenCL, SDSoC C++ and Vivado

This chapter presents the *exploratory* study for mapping an existing low-level FPGA design, of the *MatVec* kernel implemented using Xilinx Vivado, to two relatively higher-level HLS approaches SDSoC OpenCL and SDSoC C++. The study in this chapter presents first the Vivado *MatVec* design details based on a design study in [ARAM19]. The Vivado *MatVec* design is used as the reference design, in that we compare solutions developed using the SDSoC OpenCL and SDSoC C++ approaches that try to replicate the Vivado HLS design objectives, where possible. In particular, we keep the algorithm in the FPGA kernel in each solution essentially the same. This study explores three parts in each solution design which are: the **Kernel code**, the ***MatVec* IP block and the System design** and the LFRic mini-app **host code design** which invokes the *MatVec* kernel. In each part, we explore the design decisions, optimizations, and programming steps for the implementations in the three HLS approaches (Vivado, SDSoC OpenCL and SDSoC C++).

In this chapter, we also discuss what design decisions from the Vivado *MatVec* design were, or were not, possible to be replicated in the SDSoC OpenCL, and SDSoC C++ approaches. Moreover, we discuss other design options which are available in the SDSoC OpenCL and SDSoC C++ approaches. The differences in design flow methodologies between the Vivado, SDSoC OpenCL and SDSoC C++ approaches are also discussed, including how these differences affect the design decisions. This chapter focuses only on presenting the exploration design details for the *MatVec* designs in the

three HLS approaches. In particular, this chapter introduces the design options and the best performing designs will be presented in the Comparison Study in Chapter 8.

7.1 *MatVec* Reference Implementation

The *MatVec* implementation that we explore in this Chapter is based on the *reduced interconnect design* described in [ARAM19] study. This design processes 864 columns of 40-level data from the LFRic mini-app model. The design consists of twelve *MatVec* IP blocks distributed across the logic cells of a Xilinx ZU9EG FPGA. Each IP block processes a single column of data (40 double-precision floating-point matrix-vector multiplications with an 8×6 matrix). As described in Section 2.6.2 from Chapter 2, the originally provided *MatVec* code performs double-precision matrix-vector multiplication on finite element cells within an outer loop that runs over an atmospheric column of forty vertical levels in the mini-app model. This mini-app model has 864 cells, and they are distributed over six colouring groups (exposing concurrency within each group), four groups with 205 cells each plus a 32 cell group and 12 cell group. The exploration objectives in this Chapter include exploring the replication (when possible) of the Vivado *MatVec* design objectives of the *reduced interconnect design* from the [ARAM19] study. Moreover, we explore implementations having a *MatVec* design with single and multiple IP block(s), each either processing either a single cell column or multiple cell columns. The next Sections will present the exploration of the *MatVec* implementations in three HLS approaches Xilinx Vivado, SDSoC OpenCL and SDSoC C++.

7.2 *MatVec* Xilinx Vivado Design

Vivado *MatVec* design is based on the *reduced interconnect design* from [ARAM19]. In this thesis, the design from [ARAM19] is slightly modified¹ to have multiple *MatVec* IP blocks, each processing multiple of the mini-app model's cell data, one cell at a time. As described in Section 2.4 from Chapter 2, the Vivado design flow methodology consists of three stages. The following sections discuss the Vivado *MatVec* design with regard to those three stages. In the following sections we review some of the related background issues to this topic (Xilinx Vivado) before presenting the results in Chapter 8.

¹The design modification was carried out with help from the author in [ARAM19], Mike Ashworth.

7.2.1 *MatVec* Xilinx Vivado Design Overview

The main idea of the Vivado *MatVec* design is to exploit spatial parallelism on the FPGA using multiple *MatVec* IP blocks, each with its own external memory block. Figure 7.1 shows the overall Vivado *MatVec* design concept. The design depends on using the small and fast BRAM blocks as the external memory block that is attached to a *MatVec* IP block, because the BRAM block's access latency is lower than the DDR memory access latency. The ARM CPU code fills the external BRAM blocks *directly*, since in Vivado, the BRAM addresses can be mapped into the host's address space. The blocks are filled with data for several cells, ahead of the execution of the IP blocks. When the IP blocks execute, the cell data are then accessible and can be fetched from the external BRAM blocks to local BRAM elements inside the kernel IP blocks for the calculation process of each cell. A kernel thus processes a single cell at a time. The kernel in this case called a *single-cell* kernel.

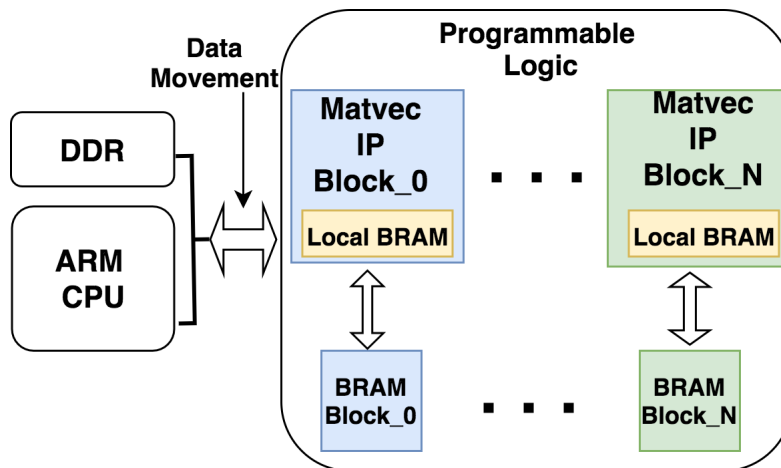


Figure 7.1: Overview of the *MatVec* Vivado HLS design.

7.2.2 *MatVec* Xilinx Vivado Kernel Code

The first stage in the Vivado design flow is writing the kernel code and generating the IP block using the Vivado HLS tool. This step is called *compile stage*, as described in the Section 2.4 from Chapter 2. Vivado HLS is also used for the analysis and synthesis of the Vivado *MatVec* kernel. It provides a synthesis report containing performance metrics and information on the applied optimizations. Using the feedback information from the Vivado HLS tool, the programmer can observe the (likely) effect of an optimization on the kernel performance without executing it.

The Vivado *MatVec* kernel code design is created based on three design objectives, see [ARAM19]. **First:** to stream the input and output data to the IP Block, targeting one 64-bit word per clock cycle. **Second:** to pipeline and overlap of the arithmetic operations targeting 64-bit multiplications and 64-bit addition operations every clock cycle. **Third:** to minimize FPGA resource usage. These objectives were carried out through several optimization decisions. The optimisation methods can be classified into three categories: **Data array access organisation** optimisations, improving the algorithm's **computation throughput** optimisations, and improving the **data movement** optimisations. These optimisation methods are applied on the starting code shown in listing 2.2 in Section 2.6.2 from Chapter 2. The resulting optimized Vivado *MatVec* kernel code is presented in Listing 7.1.

Listing 7.1: Vivado design's kernel code.

```

1  #define NDF1 8
2  #define NDF2 6
3  #define NK 40
4  #define MVTYPE double
5  #include <string.h>
6
7  int matvec_8x6x40_v6 (const MVTYPE *matrix, const MVTYPE *x,
8  MVTYPE *lhs)
9  {
10
11      #pragma HLS INTERFACE m_axi depth=128
12      port=matrix offset=slave bundle=bram \
13      num_read_outstanding=8 num_write_outstanding=8 \
14      max_read_burst_length=64 max_write_burst_length=64
15
16      #pragma HLS INTERFACE m_axi depth=128
17      port=x offset=slave bundle=bram \
18      num_read_outstanding=8 num_write_outstanding=8 \
19      max_read_burst_length=64 max_write_burst_length=64
20
21      #pragma HLS INTERFACE m_axi depth=128
22      port=lhs offset=slave bundle=bram \
23      num_read_outstanding=8 num_write_outstanding=8 \

```

```

24      max_read_burst_length=64 max_write_burst_length=64
25      #pragma HLS INTERFACE s_axilite port=return
26
27      int df,j,k;
28
29      // Local BRAM elements
30      MVTYPE ml[NDF2][NK];
31      #pragma HLS array_partition variable=ml block
32      factor=60
33      x1[NDF2][NK];
34      #pragma HLS array_partition variable=ml block
35      factor=60
36      l1[NK];
37      #pragma HLS array_partition variable=ml block
38      factor=40
39
40      //Read input data from the external BRAM
41      memcpy (x1, x, NDF2*NK*sizeof(MVTYPE));
42
43      for (df=0;df<NDF1;df++) {
44      #pragma HLS PIPELINE
45      memcpy (ml, matrix+df*NDF2*NK,NDF2*NK*sizeof(MVTYPE));
46
47      for (k=0;k<NK;k++) {
48      #pragma HLS UNROLL
49      l1[k] = 0.0;
50      }
51
52      for (j=0;j<NDF2;j++) {
53      for (k=0;k<NK;k++) {
54              #pragma HLS UNROLL
55
56              //Matrix vector multiplication
57              l1[k] = l1[k]+ x1[j][k]*ml[j][k];
58      }

```



```

59     }
60
61     // Write output data to the external BRAM
62     memcpy ( lhs+df*NK, ll , NK*sizeof(MVTYPE));
63     }
64
65     return 0;
66 }

```

The kernel code and data layout was changed from the starting code, in the following ways²:

Data array access organisation optimisations:

- The k-index loop over the vertical levels became the innermost loop by swapping loops, see line 57 in the listing 7.1. This allows sequential execution of the k-loop with potential for stride 1 access to the column of data associated with each cell, providing more computation per cell (higher granularity).
- Transposed the data array, where necessary, to support the stride 1 execution of the k-index loop.
- The kernel only computes the matrix-vector product. The update of the lhs data array is computed on the host Arm CPU.

Computation throughput optimizations:

- Unroll the innermost loops in nested loops.
- Pipeline the outer loop.
- Partition the local BRAM arrays to provide more memory access ports (and, hence, higher bandwidth). The best partitioning factors were found to be 60 for the `matrix` and `xl` arrays and 40 for `lhs` array. See lines 31, 34 and 37 in listing 7.1.

Data movement optimizations:

²The optimisations in the bullet points are undertaken from the [ARAM19] paper, except the array partitioning.

- *matrix* and *x* array data are first transferred from the DDR memory of the host to the external block BRAM storage associated with each kernel IP block. Those data are then transferred from the external BRAM block to local *BRAM_18K* logic elements in *burst mode* during the execution of the kernel.
- The *x* data array is copied to the kernel only once, to the local *BRAM_18K*, as it is constant for all the iterations of the outer *df-loop* in the processing of a cell, see line 41 in the listing 7.1.
- Slices of the *matrix* data are transferred to the local kernel *BRAM_18k* array each *df-loop* iteration, and output columns of the *lhs* data array are copied out to the external BRAM block at each iteration of the loop in *burst mode*, thus freeing local *BRAM_18k* for use in the next iteration, see line 45 in the listing 7.1.
- Upon the completion of the kernel execution, the *lhs* array is copied from the external BRAM back to the DDR memory for use by the host.

The Vivado *MatVec* kernel data movement optimizations ensure that the data streaming in and out of the IP block occurs at the rate of one word per cycle (read/write). Without these optimizations, in particular the use of burst mode, the IP block will be waiting for one read to complete before starting the next, for example. The data transfer is implemented using the `memcpy` function, which the Vivado HLS recognises and implements using “burst mode” [Xil21e].

The HLS `INTERFACE` pragmas in lines 11-25 in Listing 7.1 are used in the kernel function to define the Vivado *MatVec* IP block ports for the interconnection in the block integration stage, as described in Section 7.2.3. The kernel’s three array arguments *x*, *matrix*, *lhs*, are specified as AXI master ports (`m_axi`). These three ports in each *MatVec* IP block are connected to one (external) BRAM block port. Therefore, they are bundled together as one `m_axi_bram` port to form a single port which helps to simplify the interconnection network in the system design, as described in Section 7.2.3. The burst mode properties are defined in the HLS `INTERFACE` pragmas as follows:

- `num_read_outstanding=8`
- `max_read_burst_length=64`
- `num_write_outstanding=8`
- `max_write_burst_length=64`

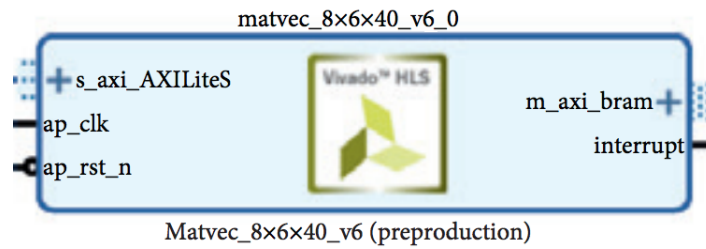


Figure 7.2: The generated *MatVec* Vivado IP block.

The `num_read/write_outstanding` specifies how many read/write requests can be made before stalling, where the `max_read_burst_length` specifies the maximum number of data to be read/written in the burst transfer.

At the end of this stage, the Vivado *MatVec* IP block is generated (as shown in Figure 7.2) and exported as an IP block that can be integrated into a full system design in the second stage of the Vivado HLS process, which is described in Section 7.2.3.

The generated Vivado *MatVec* IP block in Figure 7.2 has two ports. A Master AXI port (`m_axi_bram`) which is the result of the use of `HLS INTERFACE` pragma in lines 11-25 from 7.1. This port is the gate access for the Vivado *MatVec* IP block to the external BRAM block. The Slave AXI port (`s_axi_AXILiteS`) is a result of the use of the `HLS INTERFACE s_axilite` pragma in line 25 from 7.1. This port is the gate access of the ZynQ to the Vivado *MatVec* IP block. The Slave AXI port gives the ARM CPU an access to the control register in the generated Vivado *MatVec* IP block which is used to control the block's start/stop execution signals.

7.2.3 *MatVec* Vivado Hardware Design

The second stage (*link stage*) of the Vivado design flow is to incorporate the generated IP block into a system design that supports its execution. Therefore, the Vivado *MatVec* IP block in Figure 7.2 is incorporated into a system design using the Vivado Design Suite [Xil21f]. The Vivado system design is created by utilizing some IP blocks from the Vivado IP Catalog [Xil21g]. Those IP blocks can provide functionalities such as an interface with the ARM CPU, data handling BRAMs and clock control. The choice of the IP blocks and the interconnection method between them can be made in different ways, requiring considerable knowledge and effort to reach an appropriate design option.

Figure 7.3 shows an example with two Vivado *MatVec* IP blocks to illustrate the Vivado *MatVec* system design. The Vivado system design consist of the following IP



Figure 7.3: Overview a Vivado two *MatVec* IP blocks design.

blocks that provide the functionality of the design:

1. Twelve *MatVec* kernel IP blocks. The best found Vivado *MatVec* design consists of twelve *MatVec* IP blocks. This is the best performing Vivado design that we focus on in this Section, further details of this design's performance and other Vivado designs will be discussed in Chapter 8.
2. Twelve memory generator IP blocks to provide BRAMs per *MatVec* IP block.
3. Twelve AXI BRAM controllers to provide AXI protocol interfaces for each BRAM memory block.
4. A ZynQ UltraScale+ MPSoC IP block.
5. A clocking wizard IP block to provide a custom clock.
6. Two Reset System block processors.

The twelve Vivado *MatVec* IP blocks are replicates of the Vivado *MatVec* IP block generated in the *compile stage*. Each Vivado *MatVec* IP block is connected to an AXI BRAM controller that connects the *MatVec* block to its external BRAM memory, as illustrated in Figure 7.3.

Managing and controlling the connection between the functional IP blocks is by utilizing a set of interconnecting IP blocks and connection ports from the Vivado IP Catalog [Xil21g]. Choosing the proper connection method and IP blocks depends on the design objectives and the choice of data movement implementation. The following points summarize the interconnecting IP blocks in the Vivado system design:

1. One AXI_Interconnect IP block.
2. One AXI_Protocol_Converter IP Block.
3. Fourteen AXI_Crossbar IP blocks.
4. One Master ZynQ (HPM0) port which connects the ARM CPU with the slave ports of the *MatVec* IP blocks for ARM CPU access control on the IP blocks' control registers.
5. One Master ZynQ (HPM1) port connects the ARM CPU with the slave ports of the BRAM block controllers for ARM CPU external BRAM access.

6. A Master port on each *MatVec* IP block, see Figure 7.2, connects a slave port on each BRAM controller for external BRAM data access.

As can be seen in the example Figure 7.3, the connection between the design blocks is managed through the use of one AXI_Interconnect and fourteen AXI_Crossbar IP blocks. The AXI_Interconnect helps to do the large number of data-related conversions required, automatically converting between different clocks, different data widths, and different protocols. However, it consumes a large amount of FPGA resources. This IP block is used only to convert between the 128-bit paths of the High-Performance Master HPM ports from the ARM CPU to the 64-bit paths used in the rest of the design blocks. The AXI_Crossbar block is used for connection where the data path conversion is unnecessary, see the example Figure 7.3. The AXI_Protocol_Converter is used to explicitly convert between AXI4 on the ZynQ block and the AXI_Lite for the control register ports on the twelve Vivado *MatVec* IP blocks, see the example Figure 7.3. The two ZynQ AXI4 high-performance HPM0 and HPM1 ports are utilised to provide the ARM CPU interface with the twelve Vivado *MatVec* IP blocks. These two ports map to addresses 0xA0000000 and 0xB0000000, respectively. The addresses of the design peripherals such as BRAMs and *MatVec* IP blocks are mapped to these two address ranges to facilitate accessing them through the ARM CPU, see 7.2.4. The HPM0 port is connected to the twelve Vivado *MatVec* IP blocks' slave ports to give the ARM CPU access to the *MatVec* blocks' control registers. The other ZynQ port, HPM1, is connected to the twelve AXI BRAM controllers so that the ARM CPU can access the external BRAM blocks for data movement operations. The Vivado system design blocks' addresses can be generated automatically, but the programmer can configure them manually. The Vivado design suite provides the programmer with an address editor tool that enables the management of the system design addresses. Figure 7.4 shows the address ranges configuration for the twelve Vivado *MatVec* IP blocks and their BRAM blocks. The Vivado *MatVec* IP blocks offset addresses range started from the HPM0 ZynQ port address of 0xA0000000. The other ZynQ port, HPM1 of address 0xB0000000, is used as the base address for the external BRAM blocks' addresses range.

The Clock Wizard block used in the Vivado system design in Figure 7.4 controls multiple clock domains. The ZynQ IP block is limited to a maximum of 333MHz clock speed, but the *MatVec* IP block can run faster than that (310 MHz). Each clock domain requires a Processor System Reset block, and that is why two of them are used in the design.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
zynq_ultra_ps_e_0					
Data (40 address bits : 0x00A0000000 [256M] ,0x0400000000 [4G] ,0x1000000000 [224G] ,0x00B0000000 [256M] ,0x05...					
axi_bram_ctrl_0	S_AXI	Mem0	0x00_B000_0000	1M	0x00_B00F_FFFF
axi_bram_ctrl_1	S_AXI	Mem0	0x00_B010_0000	1M	0x00_B01F_FFFF
matvec_8x6x40_v6_0	s_axi_AXILiteS	Reg	0x00_A000_0000	8K	0x00_A000_1FFF
matvec_8x6x40_v6_1	s_axi_AXILiteS	Reg	0x00_A000_2000	8K	0x00_A000_3FFF
matvec_8x6x40_v6_0					
Data_m_axi_bram (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xB000_0000	1M	0xB00F_FFFF
matvec_8x6x40_v6_1					
Data_m_axi_bram (32 address bits : 4G)					
axi_bram_ctrl_1	S_AXI	Mem0	0xB010_0000	1M	0xB01F_FFFF

Figure 7.4: Address map example for two Vivado *MatVec* design.

Upon the completion of the Vivado *MatVec* IP blocks' integration and the configuration of the Vivado system design, the programmer directs the Vivado Design Suite to carry out the following steps to generate the bitstream of the design:

- Synthesis of the RTL-specified design into a gate-level representation.
- Placing and routing the resulting netlist onto the FPGA resources, within the physical, logical, and timing constraints of the design.
- Generation of the design Bitstream for the target FPGA configuration.

7.2.4 *MatVec* Vivado CPU code

The Vivado host CPU code is a C code that manages the addresses and the transfer from/to the DDR memory to/from the external BRAM blocks of the LFRic-mini app data arrays associated with each cell, (`x`, `matrix` and `lhs`). In addition, the host code manages the Vivado *MatVec* IP blocks execution and addresses configuration. The CPU code utilises the aforementioned ZynQ high-performance master ports, HPM0 and HPM1, to control the Vivado *MatVec* IP blocks and the external BRAM blocks. Moreover, the CPU code utilises two device tree (`/uio0` and `/uio1`) from the system device and maps them to the two zynQ ports, (HPM0 and HPM1), see code fragment in Listing 7.2 lines 11-20. These two devices are helpful to manage the addresses in the CPU code. Their contents can be checked in the device tree by using the command `dtc`, as can be seen in code fragment from Listing 7.2 lines 1-8. The two devices can be mapped into the user address space of the executable file using the `open` and `mmap` system calls, as shown in Listing 7.2 lines 22-27.

Listing 7.2: Vivado *MatVec* design's CPU code fragments.

```

1      //dtc command
2      dtc -I fs -O dts /sys/firmware/devicetree/base
3
4      //dtc command output
5      gpio@a0000000 {
6          compatible = "generic-uio", "uio";
7          reg = <0x0 0xa0000000 0x0 0x800000>;
8      };
9
10
11     //Initialise the FPGA,
12     //with settings tailored the design
13      //(0xA0000000) Matvec IP blocks
14      //base address (HPM0)
15      //(0xB0000000) External BRAM blocks
16      //base address (HPM1)
17     ierr = mvupdt_fpga_init(0, is_update ,
18         "/dev/uio0", 0xA0000000);
19     ierr = mvupdt_fpga_init(1, is_update ,
20         "/dev/uio1", 0xB0000000);
21
22      // open and mmap system calls
23     fd = open(device , O_RDWR);
24     fpgamemsize[idev] = 0x0800000;
25     fpgamemory[idev] = (char *)mmap
26         (NULL, fpgamemsize[idev], PROT_READ|
27         PROT_WRITE, MAP_SHARED, fd, 0); }
28
29      //Set up Matvec blocks addresses
30      //(Example code fragment)
31      //for two Ip blocks design
32     ierr = mvupdt_fpga_add_block(0, 0xA0000000,
33         0, 40, 1, nchunk);
34     ierr = mvupdt_fpga_add_block(0, 0xA0002000,
35         0, 40, 1, nchunk);

```



```

36
37 // Set up external BRAM blocks addresses
38 // (Example code fragment) for
39 // two BRAM blocks design
40 ierr = mvupdt_fpga_add_bram(1, 0, 0xB0000000);
41 ierr = mvupdt_fpga_add_bram(1, 1, 0xB0100000);
42
43
44 //Matvec IP Block start
45 int fpga_start (int ib_block) {
46     *control[ib_block] = 1;
47     return 0;
48 }
49
50
51 //Matvec IP Block status check
52 hold=0;
53 while((*control[ib_block]&4) == 0) {
54     hold++;
55 }
56
57 // Start addresses of arrays in BRAM
58 BRAM_matrix_start[ib] = base;
59 BRAM_x_start[ib]      = BRAM_matrix_start[ib] +
60 ncmem[ib]*nkblock[ib]*NDF1*NDF2*sizeof(MVTYPE);
61 BRAM_lhs_start[ib]    = BRAM_x_start[ib] +
62 ncmem[ib]*nkblock[ib]*NDF2*sizeof(MVTYPE);
63
64 //Set the array addresses in
65 //the block control register
66 a1 = BRAM_matrix_start[ib];
67 a2 = BRAM_x_start[ib];
68 a3 = BRAM_lhs_start[ib];
69 *(control[ib]+ic) = a1;
70 *(control[ib]+ic+2) = a2;
71 *(control[ib]+ic+4) = a3;

```

```

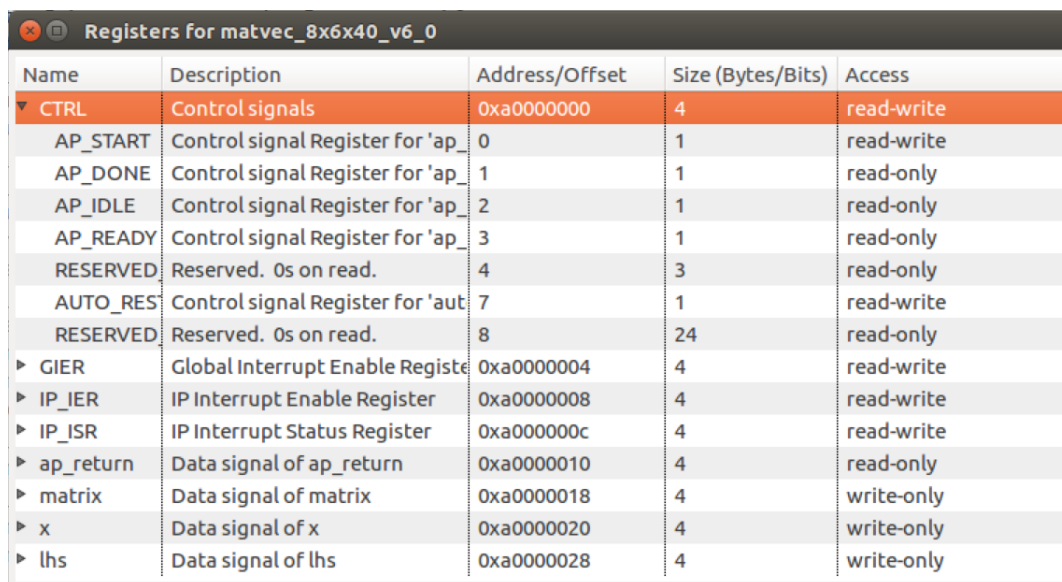
72
73 // Load the data into FPGA memory
74 // matrix
75 for (j=0;j<NDF1*NDF2*nk;j++) {
76   *(sfpga_x+j) = *(x_in+j);
77 }
78 // x
79 for (j=0;j<NDF2*nk;j++) {
80   *(sfpga_matrix+j) = *(matrix_in+j);
81 }
82
83 // get the lhs data back
84 for (j=0;j<NDF1*nk;j++) {
85   *(lhs_k+j) = *(sfpga_lhs+j);
86 }

```

MatVec and external BRAM IP blocks, addresses configuration: The two configured addresses ranges, starting at 0xA0000000 and 0xB0000000, that are mentioned in Section 7.2.3 are used in the CPU code to configure the *MatVec* IP blocks' and the external BRAM IP blocks' addresses. For example, the first Vivado *MatVec* IP block is given the base address (0xA0000000) of the HPM0 port. The second *MatVec* IP block's address is 0xA0002000 and so on for the other Vivado *MatVec* blocks, see code fragment in Listing 7.2 lines 29-35. The same addressing process is applied to the external BRAM blocks, where the first external BRAM block address is given the address 0xB0000000. The second external BRAM block's address is 0xB0100000, see code fragment in Listing 7.2 lines 37-41. The address offset between the external BRAM blocks is based on the size of the data to be stored in the BRAM block.

MatVec IP blocks, execution control: The execution process of the *MatVec* IP blocks is managed in the CPU code through the access to the *MatVec* IP blocks' control registers. Figure 7.5 shows an example set of control register details, where the first word contains control signal bits that can be configured to control the IP block's execution. For example, the AP_START bit (bit 1) can be set to "1" to start the block execution, and the AP_IDLE (bit 4) is used to check whether the Vivado *MatVec* IP block is idle. The AP_IDLE bit also can be polled to check when the IP block has finished execution. Lines 45-49 in the Vivado CPU code fragment in Listing 7.2 show the function that uses the Vivado *MatVec* control signal start bit in the control register to

start the execution of the target *MatVec* IP block. In addition, lines 51-55 in Listing 7.2 shows the function that checks the status of a target Vivado *MatVec* IP block through the use of AP_IDLE bit.



Name	Description	Address/Offset	Size (Bytes/Bits)	Access
CTRL	Control signals	0xa0000000	4	read-write
AP_START	Control signal Register for 'ap_	0	1	read-write
AP_DONE	Control signal Register for 'ap_	1	1	read-only
AP_IDLE	Control signal Register for 'ap_	2	1	read-only
AP_READY	Control signal Register for 'ap_	3	1	read-only
RESERVED	Reserved. 0s on read.	4	3	read-only
AUTO_RES	Control signal Register for 'aut	7	1	read-write
RESERVED	Reserved. 0s on read.	8	24	read-only
GIER	Global Interrupt Enable Register	0xa0000004	4	read-write
IP_IER	IP Interrupt Enable Register	0xa0000008	4	read-write
IP_ISR	IP Interrupt Status Register	0xa000000c	4	read-write
ap_return	Data signal of ap_return	0xa0000010	4	read-only
matrix	Data signal of matrix	0xa0000018	4	write-only
x	Data signal of x	0xa0000020	4	write-only
lhs	Data signal of lhs	0xa0000028	4	write-only

Figure 7.5: Example of the register map of the Vivado *MatVec* IP block.

External BRAM blocks, data addresses configuration: As mentioned previously, the ARM CPU can access the external BRAM blocks through the master HPM1 port. The address of each BRAM block in the CPU code is configured within the address range of the HPM1 port (starting at 0xB0000000). Therefore, the data addresses in each BRAM block start with that BRAM block address within the 0xB0000000 address range. The address offset between data in a BRAM block is configured based on the data size. For example, the data addressed in an external BRAM block that has a 0xB0100000 address, starts from this 0xB0100000 address. The three arrays of data associated with an IP block, (matrix, x and lhs) in this particular BRAM block, will have the following addresses. The matrix start address is the base address, 0xB0100000; the x address is 0xB0200000, the base address plus the size of matrix data; and the lhs address is the x address plus the size of the x data. This manual addressing configuration process is managed in the CPU code in lines 57-62 of Listing 7.2. This code enables the ARM CPU to write and read data from the external BRAM block directly.

To enable the *MatVec* IP blocks access to their external BRAM block data, the CPU code configures those data addresses using the IP block control registers. As mentioned earlier, each *MatVec* IP block has control registers that hold the addresses

of the three array arguments (*matrix*, *x* and *lhs*). Further down in the register map in Figure 7.5 we can see the locations of the 32-bit registers that carry the addresses of the Vivado *MatVec* IP block’s three arguments. The data addresses in the BRAM block are mapped to their corresponding control registers addresses. For example, the data address in the BRAM block with address `0xB0000000` is mapped to its attached *MatVec* IP block control registers in the following way. The register address `0xA0000018` of the *matrix* argument holds the address `0xB0000000` of the *matrix* array in that particular BRAM block. The register address `0xA0000020` of the *x* argument holds the address of the *x* data in the attached BRAM block; and the same pattern applies for the *lhs* data address. The CPU code lines 64-71 in 7.2 manage this addresses mapping configuration for each *MatVec* IP block, which enables each *MatVec* IP block to know where to find its required data in their attached external BRAM block.

In summary, the CPU code is designed to follow the following steps to execute the *MatVec* blocks, once the algorithm is started:

- Check the *MatVec* block has finished the last calculation by checking `AP_IDLE` register, see Listing 7.2 lines 51-55.
- Write data into the BRAM attached to each *MatVec* block using the appropriate address, noting the offsets for *matrix*, *x* and *lhs*, see lines 73-81 in Listing 7.2.
- Write the addresses for *matrix*, *x* and *lhs* into the control registers for each *MatVec* block, see lines 64-71 in Listing 7.2.
- Start each *MatVec* block by writing a “1” into `AP_START` register, see Listing 7.2 lines 45-49.
- Poll `AP_IDLE` register status to check the *MatVec* block to see if it has finished execution, see lines 51-55 in Listing 7.2.
- Read data from each BRAM block for the results of the computation in *lhs*, see lines 83-86 in Listing 7.2.

7.3 *MatVec* OpenCL design

This section explores the design options for creating a *MatVec* solution using the SD-SoC OpenCL approach. We explored options for replicating the Vivado kernel design objectives and optimizations where possible. In addition, we compare the *MatVec*

SDSoC OpenCL design stages against the three design stages that of the Vivado approach which were presented in Section 7.2. We also discuss the design aspects that can and cannot be replicated in the auto-compiler SDSoC OpenCL methodology when attempting to match the manual Vivado *MatVec* design.

The SDSoC OpenCL approach methodology required only two stages for creating a *MatVec* design. These two stages are writing the OpenCL kernel code and the CPU host code. The following sections explore the SDSoC OpenCL *MatVec* design with regard to those two stages.

7.3.1 *MatVec* SDSoC OpenCL Kernel code

The first stage in creating a *MatVec* design in the SDSoC OpenCL approach is writing the OpenCL kernel code. In writing the OpenCL kernel, we are following the *MatVec* Vivado kernel code optimisations and design decisions in code Listing 7.1, where applicable. This includes replicating the ***Data array access organisation*** optimisations, ***computation throughput*** optimisations, and the ***data movement*** optimisations. The Vivado *MatVec* kernel code design was created based on three design objectives (see section 7.2.2) and on the use of external BRAM blocks. The use of external BRAM blocks in the Vivado system design was a significant factor in designing the Vivado kernel code. The SDSoC OpenCL does not provide proper support for the equivalent external BRAM block creation outside of the Matvec IP block³. Therefore, the only available memory solution option in the SDSoC OpenCL approach is the use of DDR memory. In contrast to the Vivado approach, the SDSoC OpenCL methodology is a high-level HLS tool based only on inserting attributes into the kernel code. The SDSoC `xocc` compiler creates the system design automatically based on the decisions taken in writing the kernel code. Figure 7.6 shows an overview of a possible design option for the *MatVec* kernel in the SDSoC OpenCL approach. In this design, the BRAM blocks are only used as local memory buffers within the SDSoC OpenCL *MatVec* blocks (requiring only the declaration of local arrays in the kernel). The data are transferred between the host and the *MatVec* blocks through the DDR memory.

The first design decision in writing the *MatVec* SDSoC OpenCL kernel is the choice of OpenCL kernel type. As described in Section 2.3.1 from Chapter 2, two OpenCL kernel type options are available: (*task* and *NDRange* kernels). The appropriate choice for the *MatVec* kernel is the *task* kernel, since the *task* kernel proved to be a better

³Private communication with a Xilinx engineer confirms that creating on-chip global memory using BRAM is not supported.

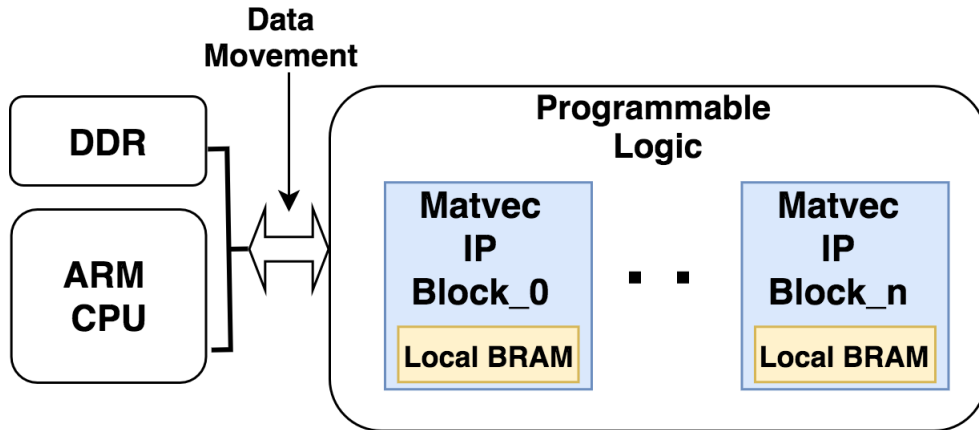


Figure 7.6: Overview of the SDSoC OpenCL *MatVec* design.

OpenCL kernel type option for FPGAs as the exploration study presented in Chapter 4 concluded. As described in the previous section, the Vivado *MatVec* kernel design reads/writes data from an external BRAM block. The external BRAM block accommodates data from several cells, and the *MatVec* IP block reads and processes only one cell at a time. In addition, the Vivado *MatVec* kernel reads slices of `matrix` data every *df* loop iteration to allow for streaming one word per cycle. BRAM access latency is lower than DDR memory access latency which was a determining factor in the kernel design decisions for the Vivado *MatVec* kernel. However, as the only memory option available in the SDSoC OpenCL approach is the use of DDR memory, some of the SDSoC OpenCL *MatVec* kernel design decisions taken have to be different to those of the Vivado kernel design. These design decisions aim to minimise the latency access overhead that can be caused by using the DDR memory. **First:** we transfer the entire `matrix` data to the local BRAM in the kernel from the DDR memory instead of using the slice-based method in the Vivado design. In addition, this data transfer occurs before the start of the *df* loop, compared to the Vivado *MatVec* kernel design. **Second:** we transfer (to the local BRAM) and process multiple cells in a single call to the kernel instead of following the cell-by-cell method used in Vivado. These two design decisions aim to minimise the access to the DDR memory as far as possible to improve the SDSoC OpenCL *MatVec* kernel latency overhead, even though it is not a replication of the Vivado *MatVec* kernel design. Choosing how many cells the SDSoC OpenCL *MatVec* can process in a call is based on two factors: the number of SDSoC OpenCL *MatVec* IP blocks to be created, and the maximum data size that the local BRAM in the IP block can accommodate. We found that the maximum number of cells that can be accommodated in only one SDSoC OpenCL *MatVec* IP block is 160

cells data. However, the more IP blocks that are created, the fewer cells the IP block can process due to resource limitations encountered when the full design is created. In Chapter 8 we evaluate and discuss SDSoC OpenCL *MatVec* designs that process both a single cell and multiple cells.

To manage the variants of cell designs in the SDSoC OpenCL *MatVec* kernel, we introduced a new loop around the *df* loop to manage the number of cells the kernel is processing. As described in Section 2.6.2 from Chapter 2, the LFRic model cubed-sphere grid has 864 cells and these are organized into six groups of colours (four 205 cells groups, one 32 and and 12 cells group). The SDSoC OpenCL *MatVec* kernel is created to be able to process the maximum number of cells that the kernel IP block can process, and we call it from the host CPU code as many times as required in order to process the number of cells in each colouring group in turn. For example, creating an SDSoC OpenCL *MatVec* kernel with 160 cells means the IP block can processes the cells of a 205 cell colouring group with two kernel calls. The first call processes 160 cells, and the second call processes the remaining 45 cells. The same idea is applied to the other colouring groups. To manage this static data scheduling⁴ in the SDSoC OpenCL *MatVec* kernel we introduced two variables. These variables are: *NCELLS* and *nchunk*. The *NCELLS* variable holds the highest possible number of cells that the IP block can process in a particular build of the kernel; the *nchunk* variable holds the number of cells that the IP block will process in a particular call. The *nchunk* variable is passed as a kernel function argument from the host code at the kernel's execution. However, the *NCELLS* variable is defined in the kernel code, because it is used to define the kernel's BRAM elements size at compile time to avoid an *xocc* compilation failure.

The relevant Vivado *MatVec* Kernel optimisation HLS pragmas are translated to the equivalent attributes in the SDSoC OpenCL *MatVec* Kernel. However, array partitioning optimisation is avoided because we found it increases the design complexity and causes a compiler error.

The creation of multiple SDSoC OpenCL *MatVec* IP blocks in the SDSoC OpenCL was achieved by replicating the kernel code in the SDSoC OpenCL *.cl* file and giving each kernel instance a different IP block name. For example, if three SDSoC OpenCL

⁴Static data scheduling means creating a kernel that processes the max number of cells regardless that it will be sometimes be called to process a smaller number of cells. This design is not an efficient solution in terms of resource usage; however, dynamic data scheduling is not suitable because the kernel cannot be compiled without the data size being known at compiler time. This issue will be looked at further in future work.

MatVec IP blocks are created, the kernel code will be written three times with different kernel function's names such as `kernel_0` to `kernel_2`.

Code listing 7.3 presents the SDSoC OpenCL *MatVec* kernel code with all the design decision changes compared to the Vivado *MatVec* kernel code that have been discussed.

Listing 7.3: SDSoC OpenCL *MatVec* kernel code

```

1 #define NDF1 8
2 #define NDF2 6
3 #define NK 40
4 #define NCELLS 160
5
6 #define MVTYPE double
7
8 __attribute__((reqd_work_group_size(1, 1, 1)))
9 __kernel void matvec_8x6x40_v6_vanilla_0 (
10     int nchunk,
11     __global MVTYPE * __restrict matrix,
12     __global MVTYPE * __restrict x,
13     __global MVTYPE * __restrict lhs)
14 {
15
16     int df,j,k,nc;
17
18     //nchunk arg is the number of cells to actually run with.
19
20     // local storage
21     MVTYPE ml[NCELLS*NDF1*NDF2*NK];
22     MVTYPE xl[NCELLS*NDF2*NK];
23     MVTYPE ll[NCELLS*NDF1*NK];
24
25     // Read x data from global memory
26
27     __attribute__((xcl_pipeline_loop(1)))
28     ix_rd: for (nc=0;nc<nchunk*NDF2*NK;nc++) {
29         xl[nc] = x[nc];

```



```

30     }
31
32     __attribute__((xcl_pipeline_loop(1)))
33     imat_rd: for (nc=0;nc<nchunk*NDF1*NDF2*NK;nc++ ) {
34         ml[nc] = matrix[nc];
35     }
36
37     for (nc=0;nc<nchunk;nc++) {
38
39         // set up base for lhs and matrix
40         int base_ll;
41         int base_xl;
42         int base_ml;
43
44         base_ll = nc*NDF1*NK;
45         base_xl = nc*NDF2*NK;
46         base_ml = nc*NDF1*NDF2*NK;
47
48
49         __attribute__((xcl_pipeline_loop))
50         df_loop: for (df=0;df<NDF1;df++) {
51
52             __attribute__((opencl_unroll_hint))
53             ll_init: for (k=0;k<NK;k++) {
54                 ll[base_ll+k] = 0.0;
55             }
56
57             matvec_j: for (j=0;j<NDF2;j++) {
58                 __attribute__((opencl_unroll_hint))
59                 matvec_k: for (k=0;k<NK;k++) {
60                     ll[base_ll+k] = ll[base_ll+k] +
61                     xl[base_xl+j*NK+k] * ml[base_ml+j*NK+k];
62                 }
63             }
64             base_ll += NK;

```

```

65             base_m1 += NDF2*NK;
66
67     } // end df loop
68
69     } // end nc loop
70
71     // Write lhs to global memory
72     __attribute__((xcl_pipeline_loop(1)))
73     lhs_w: for (nc=0;nc<nchunk*NDF1*NK;nc++) {
74         lhs[nc] = ll[nc];
75     }
76 }

```

7.3.2 *MatVec* SDSoC OpenCL, Hardware Design

In contrast to the Vivado approach, the system design creation in the SDSoC OpenCL approach for the *MatVec* kernel is processed automatically by the SDSoC OpenCL *xocc* compiler after writing the kernel code. The *xocc* compilation process is influenced by the OpenCL *MatVec* kernel code design decisions and the inserted optimisation attributes. In the compilation process, the *xocc* compiler creates the system design IP blocks, including the SDSoC OpenCL *MatVec* IP block(s), decides the interconnection methods to be used and generates the final bitstream file automatically. Figure 7.7 shows the system design generated by the SDSoC *xocc* compiler for one OpenCL *MatVec* IP block. This system design is relatively simple compared to the *MatVec* Vivado system design in Figure 7.3.

The following points explore the functional components that the *xocc* compiler created for the SDSoC OpenCL *MatVec* system design. The main generated IP blocks in Figure 7.7 are the following:

1. One OpenCL *MatVec* kernel IP block.
2. A ZynQ UltraScale+ MPSoC IP block.
3. A clocking wizard IP block to provide a custom clock.
4. One Reset System block processor.

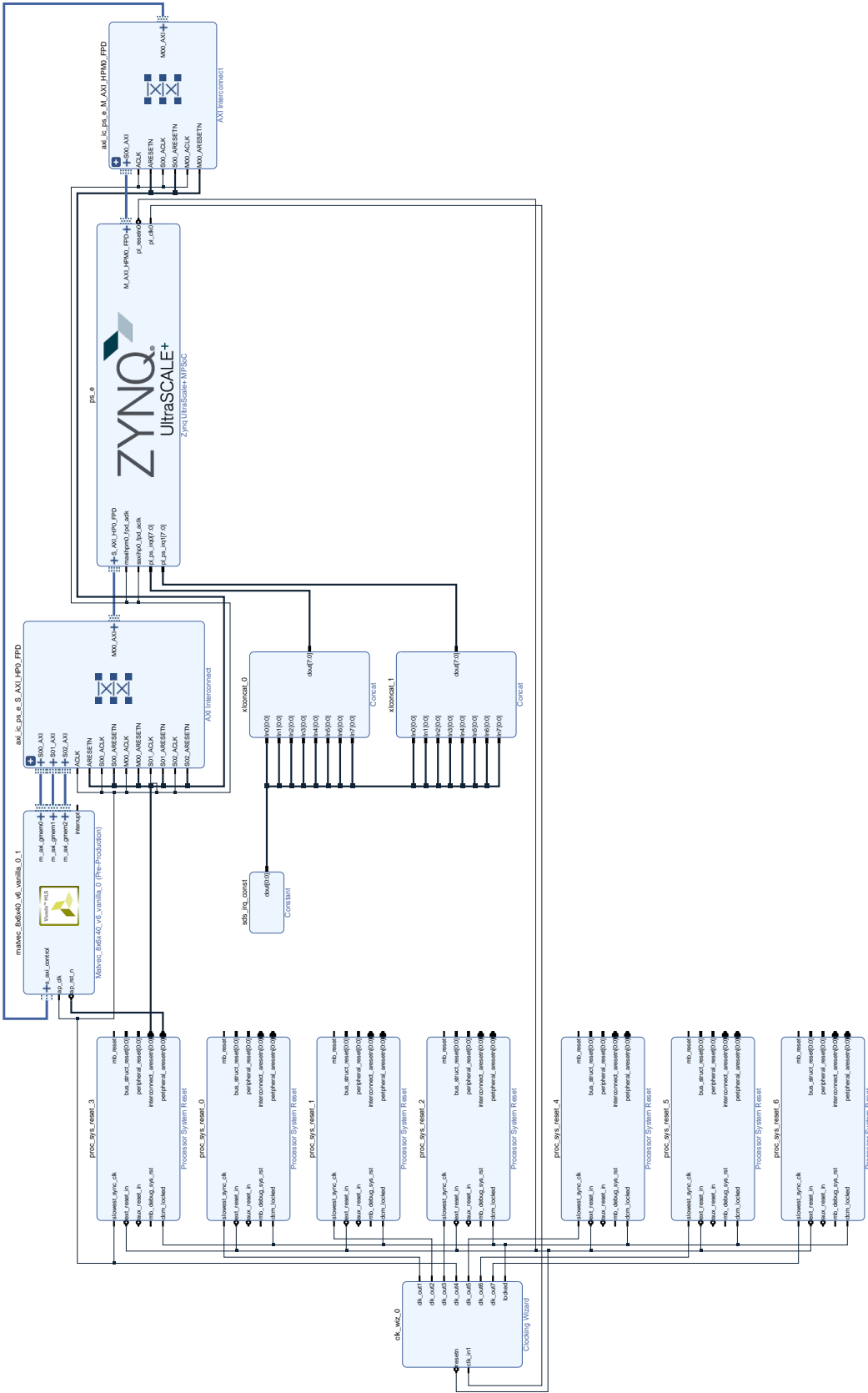


Figure 7.7: Overview of the SDSoC OpenCL one *MatVec* IP block system design

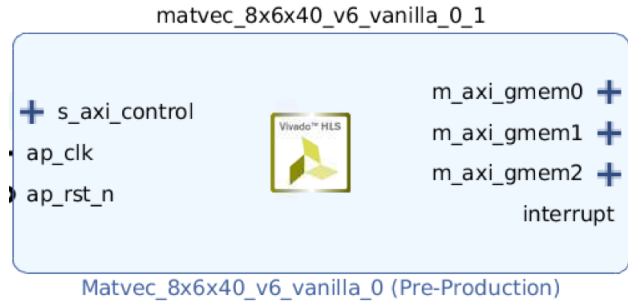
Figure 7.8: OpenCL *MatVec* IP block.

Figure 7.8 shows the generated SDSoC OpenCL *MatVec* IP block by the *xocc* compiler. The block has three Master global memory AXI ports (`m_axi_gmem(n)`), and one Slave AXI port (`s_axi_control`). The Master AXI ports are representing the three *MatVec* kernel OpenCL global memory arguments `matrix`, `x` and `lhs`. Each port gives a separate access for each of the three arguments in the DDR memory. The OpenCL *xocc* compiler generates these multi-Master AXI ports because we used the *MAX memory port* optimisation. As mentioned in Section 2.5.1 and in [ARA21b], the use of the *MAX memory port* optimisation was found to reduce DDR access time (through the increased bandwidth available).

The Slave AXI port (`s_axi_control`) in Figure 7.8, is the gate access for the ARM CPU on the SDSoC OpenCL *MatVec* IP block. This port gives the ARM CPU the with control of the IP block *control registers* to control the block execution start and stop. Unlike with the Vivado *MatVec* design approach, in the SDSoC OpenCL, the programmer is not required to interact with the *MatVec* IP block *control registers* or configure the start and stop execution bits. The kernel execution is instead handled by the OpenCL API `enqueueTask` function. This results in a large reduction in the detailed knowledge of low-level detail and effort required of the programmer.

The Clock Wizard block used in the SDSoC OpenCL *MatVec* block in Figure 7.7 controls only one clock domain. Unlike the Vivado design, where the programmer has the advantage of separating the ZynQ IP block clock speed from the Vivado *MatVec* IP blocks' clock speed. The auto-generated SDSoC OpenCL hardware design runs within the ZynQ maximum of 333 MHz clock speed. Therefore, the SDSoC OpenCL *MatVec* IP block speed cannot run faster than that 333 MHz. The highest found clock speed that the OpenCL *MatVec* IP block can run with was only 200Mhz. The OpenCL *MatVec* implementation operates on a lower clock speed (200MhZ) compared to the *MatVec* Vivado design (310MhZ), further discussion is provided in Section 7.5, and

Section 8.1.1 in Chapter 8.

The *xocc* compiler also controls automatically the generation of the interconnection method between the IP blocks in the SDSoC OpenCL *MatVec* system design, as can be seen in Figure 7.7. Two AXI_Interconnect IP blocks are generated and configured in Figure 7.7. The AXI_Interconnect IP blocks are utilised to manage the data-path conversion between the ZynQ Master HPM0 port (128-bit) and the Slave HP0 ports to/from the SDSoC OpenCL *MatVec* IP block(s).

In creating multiple SDSoC OpenCL *MatVec* IP blocks, the *xocc* compiler generates the same hardware system design as for the single SDSoC OpenCL *MatVec* IP block in Figure 7.7. However, the size of the two AXI_Interconnect blocks is increased to manage the connection of the multiple *MatVec* IP blocks' ports to the ZynQ IP block, as can be seen in Figure 7.9. The *xocc* compiler increases the number of slave ports in the AXI_Interconnect blocks to match the number of the *gmem* ports and the *s_axi_control* ports in the OpenCL *MatVec* IP blocks. This auto-generated hardware design decision can be expected to affect the performance scalability of the design, as the DDR memory bandwidth is being shared through only one AXI_Interconnect block between the four *MatVec* blocks; this will be discussed further in Section 8.1.1 in Chapter 8. In addition, the bigger the AXI_Interconnect block the more FPGA resources are consumed.

7.3.3 *MatVec* SDSoC OpenCL, Host Code

The second stage in the SDSoC OpenCL *MatVec* design approach is writing the host CPU code. Unlike the Vivado host CPU code, OpenCL provides the programmer with OpenCL API functions that hide the details of the manual address configuration, manual data transfer management and kernel execution required with Vivado. As stated in Section 2.3.1 from Chapter 2, the SDSoC OpenCL host code is responsible for the general management, the task launching associated with the execution of the *MatVec* OpenCL IP block(s) and the data transfer to/from the OpenCL global memory (DDR memory), through a set of OpenCL API functions.

The relevant host code is shown in Listing 7.4.

Listing 7.4: SDSoC OpenCL *MatVec* CPU code fragment

```
1 // Getting Xilinx Platform and its device
2 devices = xcl::get_xil_devices();
3 device = devices[0];
```

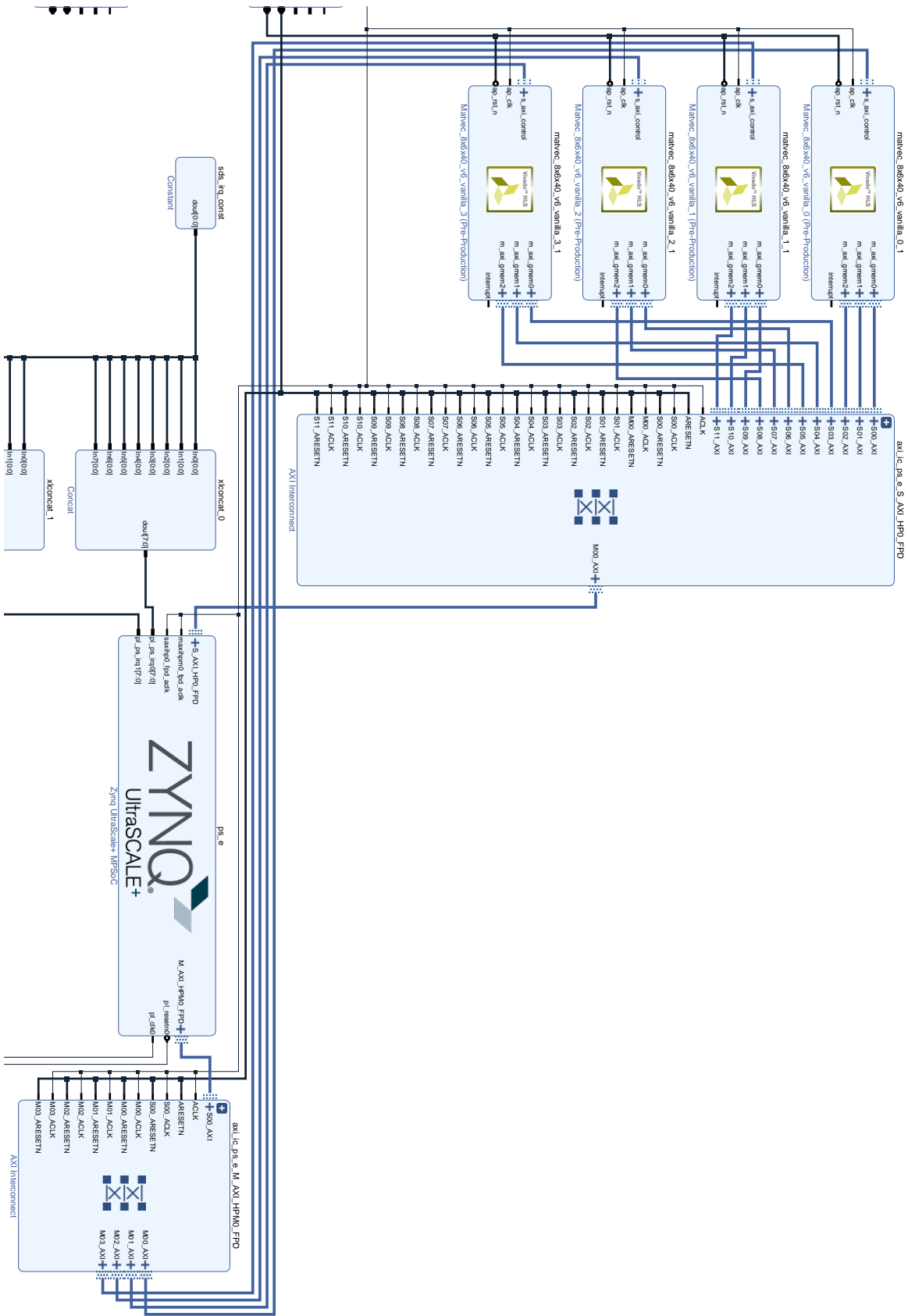


Figure 7.9: Overview of the SDSoc OpenCL four MatVec IP blocks system design

```

4  std::string device_name = device.getInfo<CL_DEVICE_NAME>();
5
6  // Creating Context
7  context = cl::Context(device);
8
9  // Creating Multiple Command Queues
10 for (int bl = 0; bl < MAX_BLOCKS; bl++) {
11   cmd_q[bl] = cl::CommandQueue(context, device);
12 }

13 // Loading XCL Bin into char buffer
14 std::string binaryFile =
15 xcl::find_binary_file(device_name, "matvec");
16 cl::Program::Binaries bins =
17 xcl::import_binary_file(binaryFile);
18
19 // Creating the program
20 program = cl::Program(context, devices, bins, NULL, &err1);
21
22 // Create array of buffers for matrix, x, lhs
23 for (int bl = 0; bl < nbreq; bl++) {
24
25   // create matrix buffers
26   buf_matrix[bl] = cl::Buffer(context, CL_MEM_READ_WRITE,
27   MAX_CELLS * ndf1 * ndf2 * nlayers * sizeof(cl_double),
28   NULL, &status);
29
30   // create x buffers
31   buf_x[bl] = cl::Buffer(context, CL_MEM_READ_WRITE,
32   MAX_CELLS * ndf2 * nlayers * sizeof(cl_double), NULL,
33   &status);
34
35   // create lhs buffers
36   buf_lhs[bl] = cl::Buffer(context, CL_MEM_READ_WRITE,
37   MAX_CELLS * ndf1 * nlayers * sizeof(cl_double), NULL,

```

```
38 &status);
39 }
40
41 // kernel_matvec: create kernel(s)
42 kernel_matvec[bl] = cl::Kernel(program, kernel_name,
43 &status);
44
45 //start the timing t0 (load time)
46 t0 = (double) clock() / CLOCKS_PER_SEC;
47
48 //write the cells data to the buffers of each IP block
49 for (bl = 0; bl < nblocks; bl++) {
50
51 //write the matrix data
52 cmd_q[bl].enqueueWriteBuffer(buf_matrix[bl], CL_TRUE, 0,
53 ncellblk[bl] * NDF1 * NDF2 * nlayers * sizeof(double),
54 mat_temp);
55
56 //write the x tata
57 cmd_q[bl].enqueueWriteBuffer(buf_x[bl], CL_TRUE, 0,
58 ncellblk[bl] * NDF2 * nlayers * sizeof(double), cpu_x);
59 }
60
61 // wait for the data transfer tasks to finish
62 for (bl = 0; bl < nblk; bl++) {
63     cmd_q[bl].finish();
64 }
65
66 // starting timing t1 (compute time)
67 t1 = (double) clock() / CLOCKS_PER_SEC;
68
69 // execute the IP block
70 //ncellblk -> number of cells the block will process
71 kernel_matvec[bl].setArg(0, ncellblk[bl]);
72 kernel_matvec[bl].setArg(1, buf_matrix[bl]);
```



```

73 kernel_matvec[bl].setArg(2, buf_x[bl]);
74 kernel_matvec[bl].setArg(3, buf_lhs[bl]);
75 cmd_q[bl].enqueueTask(kernel_matvec[bl]);
76
77 //wait for all blocks to finish execution
78 for (bl = 0; bl < nblk; bl++) {
79     cmd_q[bl].finish();
80 }
81
82 //start timing t2 (store time)
83 t2 = (double) clock() / CLOCKS_PER_SEC;
84
85 // write the lhs data back
86 // from the buffers to the host
87 cmd_q[bl].enqueueReadBuffer(buf_lhs[bl],
88 CL_TRUE, 0, ncellblk[bl] * NDF1 * nlayers *
89 sizeof(double), cpu_lhs);
90
91 //wait for the data write back to finish
92 cmd_q[bl].finish();
93
94 // take the timing t3
95 t3 = (double) clock() / CLOCKS_PER_SEC;
96 t1 = t1 - t0; (data load time)
97 tc = t2 - t1; (blocks compute time)
98 ts = t3 - t2; (data store time)

```

The following presents the *MatVec* SDSoC OpenCL host code tasks for executing the OpenCL *MatVec* design.

- Getting the Xilinx platform and finding the OpenCL device. In this case the ZCU102 FPGA board, see lines 1-4 in the code fragment in Listing 7.4.
- Create the OpenCL context and command queue for the selected device. The command queue type used in the OpenCL *MatVec* host code is an in-order command queue. We created multiple in-order command queues, each is associated to a *MatVec* IP block, see lines 9-12 in code fragment 7.4. The reason for this

choice is to execute the OpenCL *MatVec* IP blocks in parallel.

- Load the Xilinx *cl* binary file that contains the *MatVec* kernel code, see lines 13-17 in code fragment 7.4
- Create the OpenCL program, see line 20 in code fragment 7.4.
- Create OpenCL memory buffer objects in the DDR memory for the `matrix`, `x`, and `lhs` arrays. In the multiple *MatVec* IP blocks design we create an array of buffers for the `matrix`, `x`, and `lhs` arrays, each associated with a certain *MatVec* IP block, see lines 22-39 in code fragment 7.4.
- Create an array of OpenCL kernel functions and link them to the name of the *MatVec* OpenCL kernel, see lines 41-43 in code fragment 7.4.
- Transfer the data between the ARM CPU host code and the OpenCL *MatVec* IP block(s) using the OpenCL API functions *EnqueueWriteBuffer*, which copies the relevant cell data from the host DDR memory space to the OpenCL DDR memory buffers, as required, see lines 51-59 in code fragment 7.4.
- Set the kernels' argument lists and execute them in parallel, see lines 69-75 in code fragment 7.4.
- After the *MatVec* IP block(s) finish execution, the OpenCL API *EnqueueReadBuffer* function is used to read the data from the `lhs` buffers in the DDR OpenCL memory space to the host DDR memory space for further processing in the host, see lines 85-89 in code fragment 7.4.

7.4 *MatVec* SDSoC C++, design

This section explores the design options for creating a *MatVec* solution using the SDSoC C++ approach. Again, we explored options for replicating the Vivado design objectives and optimisations where possible. In addition, we compared the *MatVec* design stages in SDSoC C++ against both the Vivado and the SDSoC OpenCL design stages. Similarly to the SDSoC OpenCL approach, the SDSoC C++ methodology required only two design stages for creating a *MatVec* solution. These two stages are: writing the C++ kernel code and the CPU host code. The following subsections explore the SDSoC C++ *MatVec* design with regard to those two stages.

7.4.1 *MatVec* SDSoC C++, Kernel code

In writing the SDSoC C++ *MatVec* kernel code we followed the *MatVec* Vivado kernel code optimisations and design decisions in code Listing 7.1, where applicable. These optimisations included the **Data array access organisation** optimisations, **computation throughput** optimisations, and the **data movement** optimisations. Similarly to the SDSoC OpenCL *MatVec* design, the SDSoC C++ does not provide proper support for the equivalent external BRAM block usage in the Vivado *MatVec* design. The only memory solution option available in the SDSoC C++ approach is the DDR memory. The design creation in the SDSoC C++ approach is achieved automatically by the `sds++` compiler. The compiler decisions are influenced by HLS pragmas inserted in the kernel code. In contrast to the Vivado and the SDSoC OpenCL data-movement design, the SDSoC C++ approach requires the use of data movement engines (also called the Data Motion Network) for the movement of data between the DDR memory accessible by the FPGA and the ARM CPU, as discussed in Section 2.3.2 in Chapter 2. The Data Motion Network can be automatically generated by the *SDS++* compiler; however, the programmer is allowed to override those choices using HLS pragmas that can be inserted into the kernel code to explore solutions that may better suit the design. The main components of the Data Motion Network in the SDSoC C++ approach is the *data movers* blocks. Figure 7.10 shows an overview of the SDSoC C++ *MatVec* design. In this design, the BRAM blocks are only used as local memory elements within the SDSoC C++ *MatVec* blocks. The data travel between the host and the C++ *MatVec* blocks through the DDR memory and the DMA engines.

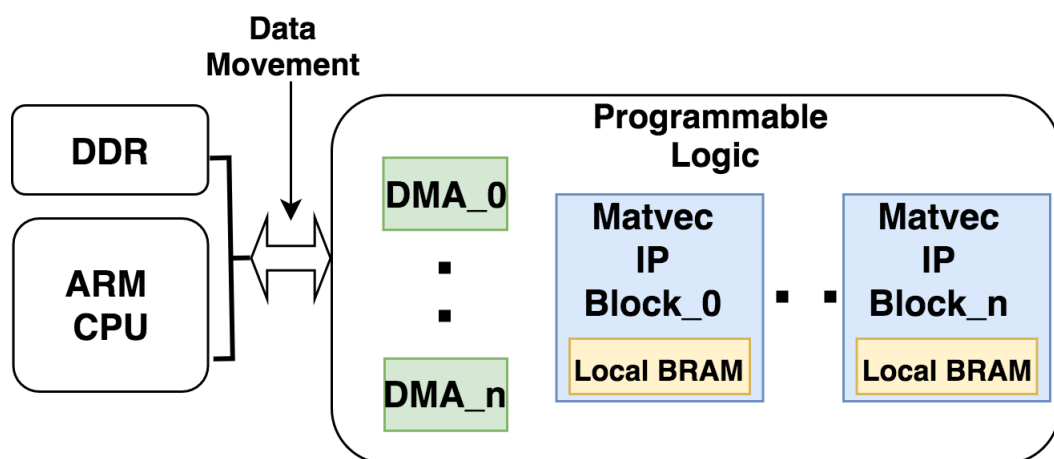


Figure 7.10: Overview of the SDSoC C++ *MatVec* design.

Similar to the SDSoC OpenCL design, we applied some design decisions in the

SDSoC C++ kernel code that are different to those of the Vivado *MatVec* kernel design. These decisions are related to using the DDR memory and aim to minimise the DDR memory access overhead. **First:** transfer the full `matrix` data to the local BRAM element instead of the slice-based method that was used in the Vivado *MatVec* kernel design. In addition, this transfer happens before the *df* loop executes. **Second:** transfer (to local BRAM elements) and process multiple cells in a single call to the kernel instead of the cell-by-cell method used in the Vivado *MatVec* kernel design. These two design choices aim to minimise the DDR memory access frequency for better overall kernel latency. As with the OpenCL design, the choice of number of cells to be processed in a call depends on the number of created SDSoC C++ *MatVec* IP blocks and on the maximum data size that the local BRAM can accommodate. We found that the maximum number of cells data that can be accommodated in only one SDSoC C++ *MatVec* IP block is 180 cells. However, the more SDSoC C++ *MatVec* IP blocks created, the fewer cells the IP block can process due to resources limitations. In Chapter 8 we evaluate and discuss SDSoC C++ *MatVec* designs that process one cell and multi-cells per call.

Similar to the SDSoC OpenCL *MatVec* kernel code, we managed the number of cells design variants in the SDSoC C++ *MatVec* kernel code by introducing a new loop around the *df* loop. In addition, we introduced the two variables *NCELLS* and *nchunk*. The *NCELLS* variable holds the highest possible number of cells that the IP block can accommodate. The *nchunk* holds the number of cells that the IP block will process in a call. The SDSoC C++ kernel is created with the maximum number of cells that the SDSoC C++ *MatVec* IP block can process, and we call it from the CPU host as many times as required to process the number of cells in an LFRic min-app colouring group.

The SDSoC C++ *MatVec* kernel code uses the same HLS optimisation pragmas utilised in the Vivado *MatVec* Kernel. However, array partitioning optimisation is avoided because we found it increases the design complexity and causes a compilation error. Creating multiple *MatVec* IP blocks in the SDSoC C++ *MatVec* kernel follows the same method as used in the SDSoC OpenCL *MatVec* kernel by replicating the kernel code in the SDSoC C++ kernel file and using different kernel function names.

In contrast to the SDSoC OpenCL and the Vivado designs, the SDSoC C++ kernel code required the use of a header file, as shown in 7.5, that defines the Data Motion Network component choice.

Listing 7.5: SDSoC C++ *MatVec* header code.

```

1  #ifndef MATVEC_KERNEL_H
2  #define MATVEC_KERNEL_H
3  #define MVTYPE double
4  #include "sds_utils.h"
5
6  #define MAX_BLOCKS 1
7  #define NCELLS 180
8  #define MAX_CELLS 180
9
10 #pragma SDS data mem_attribute(buf_matrix_0:
11 PHYSICAL_CONTIGUOUS, buf_x_0:PHYSICAL_CONTIGUOUS,
12 buf_lhs_0:PHYSICAL_CONTIGUOUS)
13
14 #pragma SDS data copy(buf_matrix_0[0:(chunk_size * 1920)],
15 buf_x_0[0:(chunk_size * 240)], buf_lhs_0[0:(chunk_size * 320)])
16
17 #pragma SDS data access_pattern(buf_matrix_0:SEQUENTIAL,
18 buf_x_0:SEQUENTIAL, buf_lhs_0:SEQUENTIAL )
19
20 #pragma SDS data data_mover(buf_matrix_0:AXIDMA_SG,
21 buf_x_0:AXIDMA_SG, buf_lhs_0:AXIDMA_SG)
22
23 void matvec_8x6x40_v6_vanilla_0(int nchunk ,
24 const double *buf_matrix_0 , const double *buf_x_0 ,
25 double *buf_lhs_0);

```

As presented in Section 2.3.2 from Chapter 2, the Data Motion Network has three components, and the critical component is the data mover engine which affects the other two component choices. We found that the most suitable DMA choice is the *AXI Scatter and Gather* (AXI_DMA_SG) data mover engine. Although AXI_DMA_SG is the slowest data mover engine option and consumes most hardware resources as discussed in Section 2.3.2 from Chapter 2, it has fewer limitations and is considered the best default sds++ compiler option. We found that the other data mover engine choices, such as AXI_DMA_SIMPLE and zero_copy, showed some limitations and caused deadlocks or compilation failure when they are used in multiple IP block designs for C++ MatVec.

Code listing 7.6 presents an example of the SDSoC C++ *MatVec* kernel code with all the design decision changes that have been discussed, compared to the Vivado and the SDSoC OpenCL *MatVec* kernels.

Listing 7.6: SDSoC C++ *MatVec* Kernel code.

```

1  #define NDF1 8
2  #define NDF2 6
3  #define NK 40
4  #define MVTYPE double
5  #include <string.h>
6
7  #include "matvec_kernel.h"
8
9
10 void matvec_8x6x40_v6_vanilla_0(int nchunk,
11 const double *buf_matrix_0, const double
12 *buf_x_0, double *buf_lhs_0) {
13
14 int df, j, k, nc;
15
16 // local storage
17 MVTYPE ml[NCELLS * NDF1 * NDF2 * NK];
18 MVTYPE x1[NCELLS * NDF2 * NK];
19 MVTYPE l1[NCELLS * NDF1 * NK];
20
21
22 //Read x data
23 ix_rd: for (nc = 0; nc < nchunk * NDF2
24 * NK; nc++) {
25 #pragma HLS PIPELINE
26     x1[nc] = buf_x_0[nc];
27 }
28
29 //Read matrix data
30 imat_rd: for (nc = 0; nc < nchunk * NDF1
31 * NDF2 * NK; nc++) {

```

```

32 #pragma HLS PIPELINE
33     ml[nc] = buf_matrix_0[nc];
34 }
35
36 for (nc = 0; nc < nchunk; nc++) {
37
38 // set up base for lhs and buf_matrix_0
39 int base_l1;
40 int base_x1;
41 int base_m1;
42
43 base_l1 = nc * NDF1 * NK;
44 base_x1 = nc * NDF2 * NK;
45 base_m1 = nc * NDF1 * NDF2 * NK;
46
47 df_loop: for (df = 0; df < NDF1; df++) {
48 #pragma HLS PIPELINE
49
50 l1_init: for (k = 0; k < NK; k++) {
51 #pragma HLS UNROLL
52 l1[base_l1 + k] = 0.0;
53 }
54
55 matvec_j: for (j = 0; j < NDF2; j++) {
56 matvec_k: for (k = 0; k < NK; k++) {
57 #pragma HLS UNROLL
58 l1[base_l1 + k] = l1[base_l1 + k]
59 + x1[base_x1 + j * NK + k]
60 * ml[base_m1 + j * NK + k];
61 }
62 }
63 base_l1 += NK;
64 base_m1 += NDF2 * NK;
65
66
67

```

```

68 } // end df loop
69
70 } // end nc loop
71
72 lhs_w: for (nc = 0; nc < nchunk * NDF1
73 * NK; nc++) {
74 #pragma HLS PIPELINE
75     buf_lhs_0[nc] = ll[nc];
76 }
77
78 } //end of kernel

```

7.4.2 *MatVec* SDSoC C++, Hardware Design

The SDSoC C++ *sds++* compiler generates the *MatVec* IP block and the system design automatically, as compared to the manual design generation required in Vivado. Similar to the SDSoC OpenCL approach, the *sds++* compiler system design creation is influenced by the kernel code design decisions and the (HLS) pragmas inserted by the programmer. Figure 7.11 shows the generated system design with a single SDSoC C++ *MatVec* IP block.

This Figure shows that the SDSoC C++ *MatVec* system design is more complicated than that of both the Vivado and the SDSoC OpenCL system designs, mainly because of the utilisation of the SDSoC C++ data motion network components.

The following points explore the SDSoC C++ *MatVec* system design. The main generated IP blocks in Figure 7.11 are the following:

1. One *MatVec* kernel IP block.
2. A ZynQ UltraScale+ MPSoC IP block.
3. A clocking wizard IP block to provide a custom clock.
4. One Reset System block processor.

Figure 7.12 shows the auto-generated SDSoC C++ *MatVec* IP block. It has three buffer ports for the `matrix`, `x` and `lhs` arrays, and one Slave AXI port (`s_axi_control`). In contrast to the SDSoC OpenCL and Vivado *MatVec* IP blocks, there is no use of Master AXI memory ports in the C++ *MatVec* IP block. The interface of the three

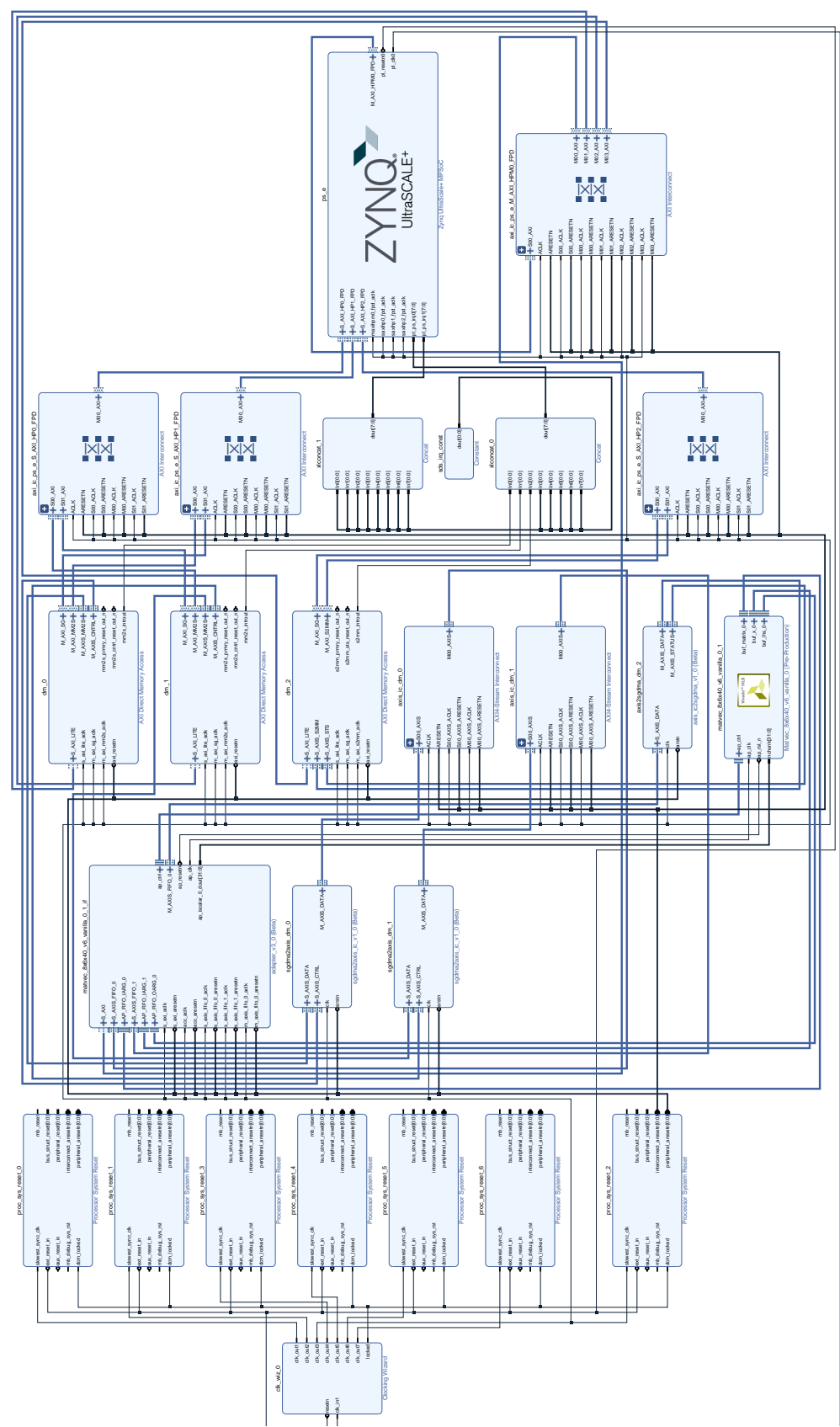


Figure 7.11: Overview of the SDSoC C++ one *MatVec* IP block system design

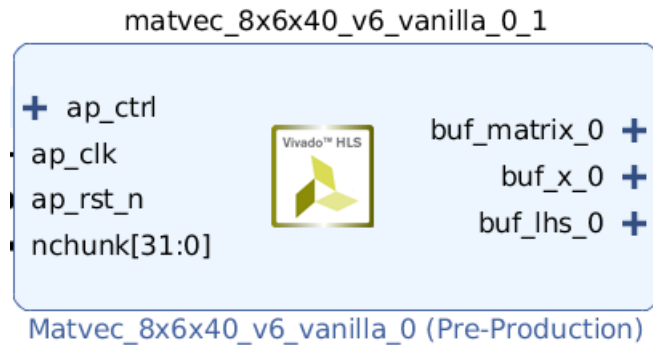


Figure 7.12: The SDSoC C++ *MatVec* generated IP block.

buffer ports in Figure 7.12 with the DDR memory is managed through the use of a C++ Data Motion network. The ARM CPU utilises the `s_axi_control` port in the SDSoC C++ *MatVec* IP block to control its execution. Similar to the SDSoC OpenCL design, the Clock Wizard block used in the SDSoC C++ design controls only one clock domain and runs within the ZynQ maximum of 333 MHz clock speed. We found that the highest clock speed that the SDSoC C++ *MatVec* IP block can run at is only 150 MHz. This SDSoC C++ *MatVec* implementation clock speed (150MHz) is slower compared to the *MatVec* Vivado design (310MHz), further discussion is provided in Section 7.5, and in Section 8.1.1 in Chapter 8.

Figure 7.11 shows the generated Data Motion Network components. As described in Section 2.3.2 from Chapter 2, a Data Motion Network consist of three components. As can be seen in that figure, those components are: the ZynQ interface ports (A), the data mover (B) and the accelerator interface ports (C). The following points explore the corresponding selections of those three components in the generated SDSoC C++ system design in Figure 7.11.

The first Data Motion Network component generated is the **ZynQ interface port** (A). The SDS++ compiler generated three ZynQ memory interface ports (`S_AXI_HPn_FPDn`) on the ZynQ block to support the connection between the data mover engines and the DDR memory in the ZynQ block. As stated in Section 2.3.2 and Section 3.1, the ZynQ block provides two types of ZynQ interface ports: a cache-coherent (`S_AXI_HPCn_FPD`) interface port, and a non-cache coherent (`S_AXI_HPn_FPD`) interface port. The SDS++ compiler default choice is the (`S_AXI_HPn_FPD`) interface port. This interface choice is a suitable choice for the *MatVec* design because it support the utilisation of the four DDR memory ports. In the case of creating multiple C++ *MatVec* IP blocks, the four `S_AXI_HPn_FPD` interface ports are shared between the created data mover engines, as can be seen in Figure B.1 in Appendix B.

In addition, the DDR memory bandwidth in Figure B.1 in Appendix B is shared by multiple data mover engines which would limit the design performance. The data width in the S_AXI_HPn_FPD port is 128-bit, so the SDS++ compiler utilised an AXI_Interconnect IP block per a ZynQ interface port to convert between the data movers' 64-bit data paths and the 128-bit paths of the S_AXI_HPn_FPD ports.

The second Data Motion Network component generated is the **Data mover component (B)**. Since our DMA engine choice in Listing 7.5 was SG, the sds++ compiler generated multiple Scatter and Gather (AXI_DMA_SG) data mover engines with multiple AXI Direct Memory Access IP blocks to support the data movement between the MatVec IP block and the ARM CPU, as can be seen in Figure 7.11.

The third Data Motion Network component generated is the **Accelerator interface port (C)**. The SDS++ compiler generated multiple AXI4_Stream Interconnect IP blocks to support the connection between the C++ MatVec IP block ports and the Scatter and Gather (AXI_DMA_SG) DMA engines. In addition, one adapter_v3_0 IP block was generated to support the connection between the C++ MatVec IP block ports and the FIFO ports. As stated in Section 2.3.2, specifying the accelerator interface component (C) depends on the accelerator's argument types. The C++ MatVec IP block's arguments types are arrays. Therefore, two accelerator interface options were available, either RAM interface (for Random data access) or streaming interface (for Sequential data access). The MatVec data is organised in the memory in a sequential manner, so we found that the most suitable accelerator interface in this case was the streaming interface. In Listing header code 7.5 (see lines 17-18) we have used the SDS data access pattern pragma to guide the sds++ compiler to use the streaming interface option. As a result, Figure 7.11 shows that the compiler introduced an adapter IP block to define the C++ MatVec IP block buffer ports as FIFO ports, which enables the data streaming behaviour. In addition, this pragma guides the sds++ compiler to create multiple AXI4_Stream Interconnect IP blocks which manage this data streaming to the C++ MatVec IP block.

Figure 7.11 shows that in a single SDSoc C++ MatVec block design, the sds++ compiler utilised three AXI_Interconnect IP blocks, three AXI_DMA_SG with three AXI Direct Memory Access IP blocks and two AXI4_Stream Interconnect IP blocks. However, in multiple SDSoc C++ MatVec IP block designs, the number of the generated IP blocks increased with regard to the number of C++ MatVec IP blocks created. This resulted in: four AXI_Interconnect blocks (one per ZynQ interface port); Multiple AXI_DMA_SG, AXI Direct Memory Access and AXI4_Stream

Interconnect IP blocks, as can be seen in Figure 7.9, showing a four *MatVec* IP block design, and Figure B.1 in Appendix B showing a six *MatVec* IP block design.

Moreover, the *SDS++* compiler increased the number of `adapter_v3_0` IP blocks to provide *FIFO* ports that match the number of created C++ *MatVec* IP blocks, again, which can be seen in Figure 7.9 and B.1 in Appendix B.

7.4.3 *MatVec* C++ Host Code

The second stage in the SDSoC C++ *MatVec* design approach is writing the host CPU code. The host code in the SDSoC C++ approach is responsible for allocating and populating buffers, executing kernels and reading the data back from the C++ *MatVec* IP blocks. The manual addresses manipulation and kernel execution are abstracted in this approach compared to the Vivado design approach.

Listing 7.7 shows the code fragment for the C++ *MatVec* host code.

Listing 7.7: SDSoC C++ *MatVec* host code fragment.

```

1  // Create buffers for the block(s) arguments
2  //_0 indicate the first IP block. higher number is the
3  // next IP block
4
5  buf_matrix_0 = (double*) sds_alloc(
6  MAX_CELLS * ndf1 * ndf2 * nlayers * sizeof(double));
7
8  buf_x_0 = (double*) sds_alloc(MAX_CELLS * ndf2
9  * nlayers * sizeof(double));
10
11 buf_lhs_0 = (double*) sds_alloc(MAX_CELLS * ndf1
12 * nlayers * sizeof(double));
13
14 // timing t0 (start load)
15 t0 = (double) clock() / CLOCKS_PER_SEC;
16
17
18 //the first IP block (0) data load
19 if (b1 == 0) {
20 for (j = 0; j < ncellblk[b1] * NDF1 * NDF2 * nlayers; j++) {

```

```

21      buf_matrix_0[j] = mat_temp[j];
22  }
23
24  for (j = 0; j < ncellblk[b1] * NDF2 * nlayers; j++) {
25      buf_x_0[j] = cpu_x[j];
26  }
27
28  for (j = 0; j < ncellblk[b1] * NDF1 * nlayers; j++) {
29  buf_lhs_0[j] = cpu_lhs[j];
30  }
31
32  }
33
34  //timing t1 (start compute)
35  t1 = (double) clock() / CLOCKS_PER_SEC;
36
37  // execute the the first IP block (0)
38  matvec_8x6x40_v6_vanilla_0(chunk_size, buf_matrix_0, buf_x_0,
39                               buf_lhs_0);
40
41  //timing t2 (start store)
42  t2 = (double) clock() / CLOCKS_PER_SEC;
43
44  //the first IP block (0) data store
45  if (b1 == 0) {
46  for (j = 0; j < ncellblk[b1] * NDF1 * nlayers; j++) {
47  cpu_lhs[j] = buf_lhs_0[j];
48      }
49  }
50
51  // timing t3 (finish)
52  t3 = (double) clock() / CLOCKS_PER_SEC;
53  t1 = t1 - t0; // load time
54  tc = t2 - t1; // compute time
55  ts = t3 - t2; // store time

```

The following points describe the host code tasks for executing the SDSoC C++ *MatVec* design:

- Create and allocate multiple buffers that hold the cells data with the use of the SDSoC `sds_alloc` function for each created C++ *MatVec* IP block. This function ensures that the data are allocated contiguously in the DDR memory, see lines 5-12 in code fragment 7.7.
- Load the buffers with cell data, see lines 18-32 in code fragment 7.7.
- Execute the C++ *MatVec* IP block(s), see line 38 in code fragment 7.7.
- After the C++ *MatVec* IP block(s) finish executing, the output data are transferred from the `lhs` buffers to the host array, see lines 44-48 in code fragment 7.7.

7.4.4 Other SDSoC OpenCL and C++ *MatVec* Design Alternatives

In this section, two design options are explored for minimizing the overhead of accessing the DDR memory in the SDSoC OpenCL and C++ implementations. In addition, there is an outstanding issue that needs to be addressed: that is that timing the SDSoC OpenCL and the SDSoC C++ kernels pure computational time (compute time) is not feasible. The utilization of an external BRAM block in the Vivado *MatVec* design allows for timing the *MatVec* pure computational time (compute time) separately from the data-movement time between the ARM CPU and the external BRAM block⁵. Timing the SDSoC OpenCL and the SDSoC C++ *MatVec* kernels compute time cannot be separated out from the data-movement time because of the use of DDR memory by the kernels. Therefore, this section explores two design options. The first option is the use of the Dataflow method to minimize the DDR memory access overhead by overlapping the data movement and the kernel computation. The second design option is the use of `pipes`, which theoretically can allow for separating the data-movement timing from the kernel's pure compute time. Both methods, in principle, allow overlapping between the kernel execution and the data movement, promising solutions for minimizing the DDR memory access overhead. Moreover, the `pipes` design method can allow for creating three kernels (a Read kernel, a Compute kernel and a Write kernel) that can be timed separately. Our interest in timing only pure compute time in the SDSoC OpenCL, and C++ is to enable a more direct comparison against the Vivado pure compute and data movement times.

⁵See the Vivado timing detail in section 8.1.2 from Chapter 8.

Dataflow Method

The Dataflow method is useful on a set of sequential tasks where there is the possibility of pipelining (overlapping) the tasks' execution [Xil21e], such as with the three SDSoC OpenCL and C++ *MatVec* kernel tasks: *load* data from DDR memory to the local BRAM; *compute* operations; and *store* data back to the DDR memory. The utilisation of the Dataflow method pipelines those tasks which potentially results in data movement and calculation overlapping. In addition, the use of Dataflow can be used to execute independent tasks in parallel.

We explored the use of dataflow in the SDSoC OpenCL and C++ *MatVec* designs by creating four functions in the kernel code that represent the three *MatVec* tasks, (*load*, which has two functions, *compute* and *store*, with one function each), as can be seen in code listing 7.8.

Listing 7.8: OpenCL *MatVec* Dataflow kernel function code fragment

```

1  // OpenCL Matvec Kernel
2  __kernel void __attribute
3  __((reqd_work_group_size(1, 1, 1)))
4  __attribute__ ((xcl_dataflow))
5  matvec_8x6x40_v6_vanilla_0 (
6      int nchunk,
7      __global MVTYPE * __restrict matrix,
8      __global MVTYPE * __restrict x,
9      __global MVTYPE * __restrict lhs)
10 {
11
12 // local storage
13 MVTYPE ml[NCELLS*NDF1*NDF2*NK];
14 MVTYPE xl[NCELLS*NDF1*NDF2*NK];
15 MVTYPE ll[NCELLS*NDF1*NK];
16
17 //input data functions
18 read_ml(matrix, ml, nchunk);
19 read_xl(x, xl, nchunk);
20
21 //calc function

```

```

22 cal(ml, xl, ll, nchunk);
23
24 //output data function
25 write_ll(lhs, ll, nchunk);
26
27 }

```

Using Dataflow aims to overlap the execution of these three tasks; ideally, a kernel's data movement time overlaps with the compute time to hide the DDR data access latency and thus increase the kernel's performance. The utilisation of Dataflow was tested in a design using one *MatVec* block with 26 cells⁶ for both the SDSoC OpenCL and the C++ approaches.

Table 7.1 shows details of latency figures for the SDSoC OpenCL 1 block, 26 cells Dataflow test design, which we discuss first.

Table 7.1: Latency figures comparison between the SDSoC OpenCL *MatVec* Dataflow implementation Versus the SDSoC OpenCL *MatVec* design with no Dataflow in terms of Latency (clock cycles) figures. The test design is One *MatVec* block with 26 cells.

Implementation Name	Load Matrix Latency	Load x Latency	Calc Latency	Store lhs Latency	Total Latency
1B_26C_NO_DF	6241	781	3145	1041	11208
1B_26C_DF	50057	78625	49932	8455	78626

For comparison purposes, Table 7.1 shows two designs, one using the Dataflow method (*1B_26C_DF*) and the other not (*1B_26C_NO_DF*). The latency figures for the *MatVec* design with Dataflow (*1B_26C_DF*) shows an overlapping between the four functions that execute the three tasks: *load*, *compute* and *store*. The overlap is apparent because the overall latency is equal to the latency of reading the *x* array data, which is the task with the highest latency. Although an overlapping has been successful between the three *MatVec* tasks, the results show that the use of the Dataflow method has shown no performance benefit compared to the non-Dataflow implementation. The latency of the *1B_26C_NO_DF* implementation (11208 CC) is much lower than that of the *1B_26C_DF* implementation with 78626 CC. The difference in the latency figures between the two designs results from the following three reasons. Firstly, to comply

⁶We based this exploration on the SDSoC OpenCL and C++ versions that delivered the best overall time, see section 8.1.2 from Chapter 8.

with the Dataflow design style, we had to expand the x array data size to allow the computation to flow, see line 14 in Listing 7.8. The x array data is replicated eight times x to match the ml data size, which allows the computations in the *compute* function to flow. Secondly, the timeline reports for each version that the SDSoC system produces show that the no-dataflow design allows the *xocc* compiler to utilize a wider port data width than that utilized in the dataflow design. In the *1B_26C_NO_DF* design, the reports show that the *xocc* compiler utilized 512 bit width read/write operations compared to only 64-bit width read/write operations in the *1B_26C_DF* design. These first two reasons explain the significant increase in the ml , x and lhs latency for the dataflow version compared to the latency figures in the no-dataflow design in Table 7.1.

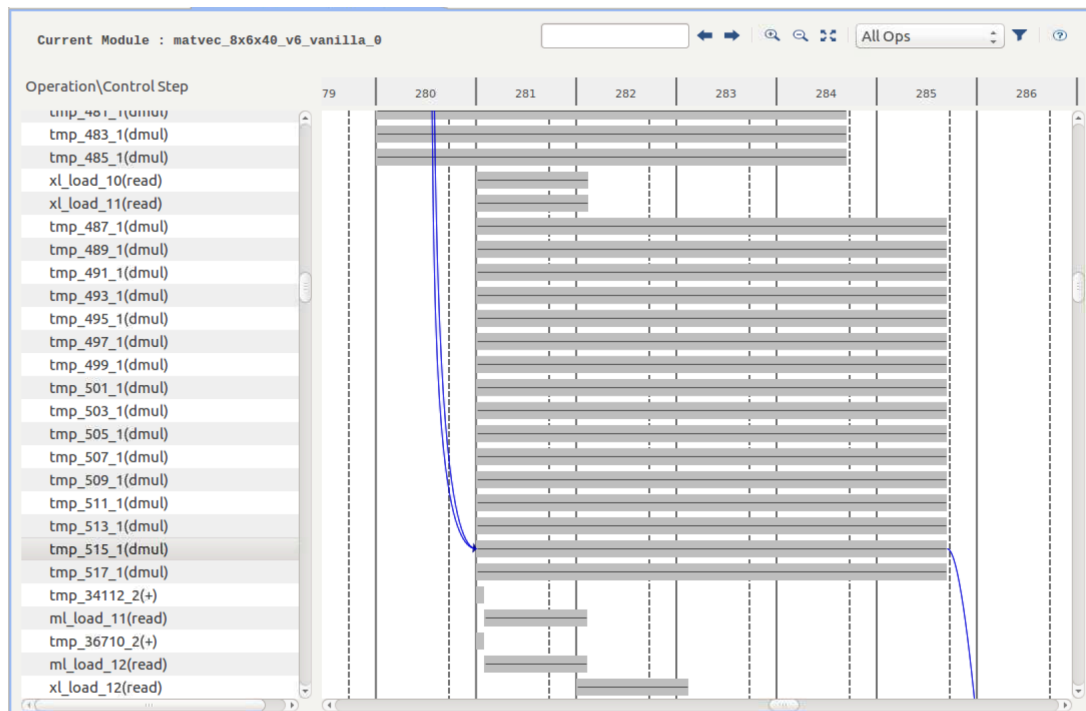


Figure 7.13: The flops timeline of the Calc function in the SDSoC OpenCL *MatVec* no-Dataflow implementation.

Table 7.2: Calc function latency breakdown of the SDSoC OpenCL *MatVec* Dataflow implementation Versus the SDSoC OpenCL *MatVec* design with no Dataflow.

Implementation Name	Total Latency	Iteration Latency	Initiation Interval	Trip Count
1B_26C_NO_DF	3145	41	15	208
1B_26C_DF	49932	251	240	208

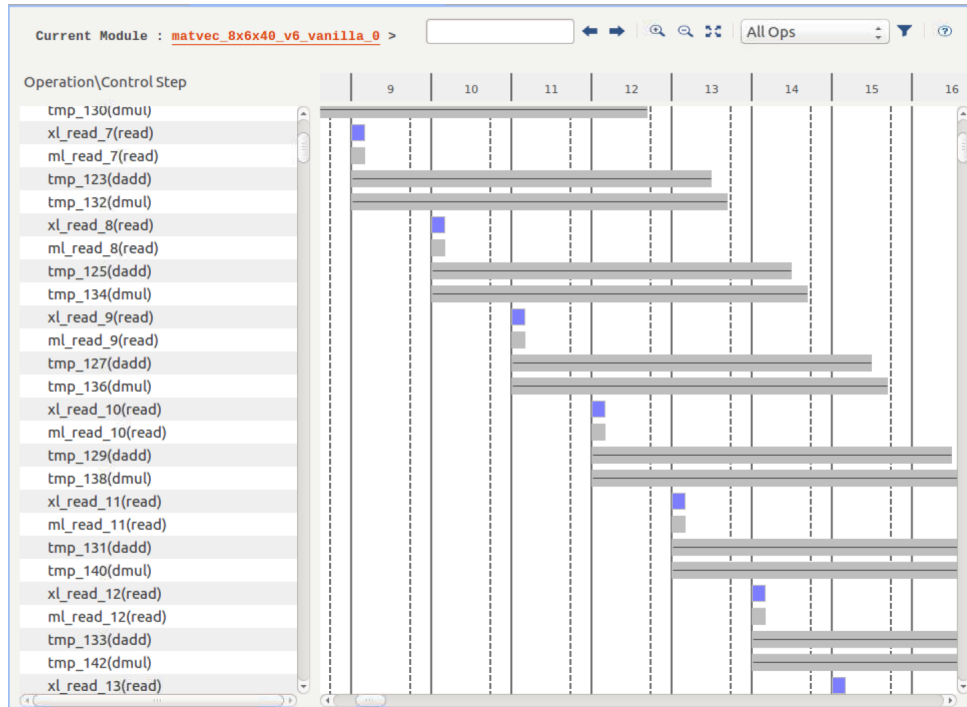


Figure 7.14: The flops timeline of the Calc function in the SDSoc OpenCL *MatVec* Dataflow implementation.

The third reason is related to the number of executed flops per cycle in the *compute* function (Calc) in each version. Figure 7.13 and Figure 7.14 show the timeline of the Calc function in the no-Dataflow and Dataflow implementations, respectively. The timeline shows that the internal computation in the dataflow design executed only 2 flops per cycle (see Figure 7.14) compared to the 16 flops per cycle in the non-dataflow design (see Figure 7.13). Table 7.2 presents a latency breakdown of the Calc function in the no-Dataflow and Dataflow implementations. The initiation interval of the Dataflow design is 16x (240), bigger than the no-Dataflow design (15). In addition, the cost of executing one iteration in the Dataflow design is 6.12x (251 CC) times higher than the no-Dataflow iteration cost (41 CC). This evidence supports the explanation of the significant differences in the latency figures from the use of the Dataflow design style compared with those of no-dataflow style in the *MatVec* kernel.

A similar exploration was undertaken on the SDSoc C++ *MatVec* Dataflow design, and we found a similar conclusion: that the benefit of the use of the Dataflow method is limited, as described above, with the *MatVec* kernel.

Multi-kernels with pipes Design

As we discussed in Section 2.5.1 from Chapter 2 the only version of pipes mode that is supported in the SDSoC approach is the blocking mode. That means a pipe has to be filled before the next kernel can start using that pipe. This design style is based on creating three kernels: a *load* kernel, a *compute* kernel and a *store* kernel. These kernels communicate through FIFO-based pipes. In the SDSoC OpenCL approach we used the OpenCL *pipes* mechanism [Xil21c] and in the SDSoC C++ approach we used the `hls::stream` pragma [Winc], which provides similar functionality. The *Load* kernel is responsible for filling two FIFOs (one for the `matrix` array and one for the `x` array) with the cell data from the DDR memory. The *compute* kernel reads from these FIFOs, and the *store* kernel reads data from the *compute* kernel through FIFOs and writes it back to the DDR memory. This design style aims to overlap the kernel computation and the data movement and enable separate timing of the compute phase from the data movement phases to support a direct comparison against the Vivado pure compute time. However, for both the SDSoC OpenCL and C++ implementations, the compute time has shown no improvement with the use of pipes as the performance was limited by the use of blocking pipes which, essentially, serializes the access to the data.

7.5 Summary of *MatVec* Exploration Study

This chapter explored the mapping of the Vivado *MatVec* design to the higher-level HLS approaches SDSoC OpenCL and the SDSoC C++. We have explored what can and cannot be replicated from the Vivado design optimisations and methods to the SDSoC OpenCL and the SDSoC C++ *MatVec* design solutions. In addition, alternative design options were attempted regarding the available design methods in the SDSoC OpenCL and C++ approaches. This Chapter also discussed and compared other possible *MatVec* SDSoC OpenCL and C++ designs. In addition, the design differences between the three approaches and the design justifications were discussed.

This exploration study shows that some Vivado *MatVec* design aspects were mapped successfully in the SDSoC OpenCL and the SDSoC C++ *MatVec* designs. However, other design choices did not, or could not, be applied to the SDSoC OpenCL and the SDSoC C++ *MatVec* designs. The manual design manipulation in the Vivado approach allows for more design freedom and options compared to the SDSoC OpenCL and C++ approaches. Two main Vivado design methods were not able to be replicated in the SDSoC OpenCL and C++ designs. The creation and use of external BRAM

blocks outside the *MatVec* IP blocks and the ability to have two clock domains being two of the most significant. The SDSoC OpenCL and SDSoC C++ designs, therefore, used DDR memory and only one clock domain. The use of DDR memory influenced how the SDSoC OpenCL and SDSoC C++ kernels are designed. We have made some necessary kernel design changes that are different to the Vivado *MatVec* kernel code. Those changes were that the SDSoC OpenCL and C++ *MatVec* IP blocks process multiple data cells in a single kernel call, and the `matrix` data is transferred into the BRAM in the kernel IP block in full, rather than by slice with Vivado. Therefore, the Vivado *MatVec* kernel code is slightly different to the SDSoC OpenCL and the SDSoC C++ kernel codes.

The Vivado *MatVec* IP block runs with a 310 MHz clock frequency because the Vivado *MatVec* system design has two separated clock domains. In contrast, the SDSoC OpenCL design runs at only 200 MHz, and the SDSoC C++ design runs at 150 MHz which are found to be the maximum.

In terms of programmability, This study shows that the Vivado approach is rather traditional high-level programmer unfriendly as it required the highest amount of low-level hardware and Vivado tool knowledge and design experience to work through the many implementation options and design trade-offs that are available, as well as development effort. This is because the approach requires the programmer to take care of many low-level concerns to produce an FPGA solution, including the manual creation of the FPGA hardware system design which requires the explicit configuration of the data-movement and connections between the IP blocks, address manipulation, setting widths of data paths and managing the execution of the kernel IP blocks through setting and examining the start and stop bits of the kernel. In addition, the programmer has to design and implement the management of the preparation and transfer of data to the FPGA BRAM blocks explicitly in the host code. This all requires significant effort. The writing and optimization of kernel code requires a similar level of effort to other methods, however, essentially, only the syntax of the pragmas required for pipelining and unrolling, etc. change. In contrast, the OpenCL and C++ approaches hide most of the complexity of the system design steps required by the Vivado approach and assign it to the SDSoC compilers. In these two approaches, the programmer needs only to know the SDSoC design flow and to have a good understanding of the kernel code pragmas for pipelining and unrolling, etc., and knowledge of their effect on the performance. The C++ approach requires also an understanding of the data-movement engines available in the system and their advantages and disadvantages. However, the

resulting automatic designs, while generally more simple, are not as efficient as can be achieved manually with Vivado. This is presumably because of the generality of the design solution, but this generality can lead to issues with timing in designs, limiting the maximum clock rate that can be used, for example.

7.6 Summary of the Two Exploratory Studies

This section summarises the key findings from the two conducted exploratory studies on the two benchmarks, SWM and MatVec.

7.6.1 Single kernel

The first exploratory study (part one) of mapping the independent operations in the L100 kernel in Chapter 4 concluded that the use of Data-Flow over functions mechanism provided the best mapping option of the kernel's concurrency. In both SDSoC OpenCL and Vivado approaches, Data-Flow over functions mechanism showed better execution time over the other mapping mechanisms. However, this mechanism required higher effort in programmability and resource consumption than the other design options.

Processing the kernel's operations in parallel, where possible, was found to be the appropriate method to achieve good computational performance if the kernel's operations are independent. This proved to be the appropriate design option for the L100 kernel from the SWM benchmark. However, utilizing this design method for the MatVec kernel calculations from the LFRic benchmark, in the second exploratory study from Chapter 7, showed no performance benefit compared to the other design options. The use of Dataflow with the MatVec kernel has an impact on the kernel's latency (high latency), limits the read/write operations bit-width to 64 bit, and reduces the number of flops per cycle that the kernel computations can produce.

7.6.2 Multiple kernels

Mapping multiple kernels was explored in Chapter 5 (different Multiple SWM kernels) and Chapter 7 (Multiple MatVec kernels). These two studies explored the effect of the selected mapping mechanisms as the application problem size changed and as the number of implemented kernels varied. In addition, the kernel-to-kernel communications options available in the HLS approaches used were explored and their effects

on performance, resource usage and programmability were investigated.

The results of mapping the multiple kernels in SWM using both the SDSoC OpenCL and the Vivado approaches showed that the kernel optimization level severely limited the problem size and, hence, the number of kernels that a user design could implement. Five out of nine SWM kernels were able to be implemented with a maximum problem size of 53×53 being possible with the selected kernel design style. In the SDSoC OpenCL implementations, three SWM kernels (L100, L200 and L300) were designed using the Data-Flow style and the two SWM kernels (L100pc and L200pc) were implemented using the Pipeline design style. In the five SWM kernels Vivado implementation, the Pipeline design style was found to be the appropriate mapping mechanism, although Data-Flow provided better performance in the single L100 Vivado mapping exploration. Utilising the Data-Flow mechanism with the multiple kernels efficiently depends on the memory design solution selected. The use of DDR memory in the OpenCL design helped provide enough memory bandwidth with multiple Data-Flow kernels compared to the Vivado multiple Data-Flow kernels solution which used BRAM external to the kernels to share data. The use of an external BRAM solution in the Vivado design was not a suitable choice for Data-Flow-style kernels as it requires extensive use of BRAM resources and increases the design complexity, particularly in terms of the manual addressing and data management required. This latter effort is much less when using the pipeline style to communicate between the kernels; this is a good example of the kind of trade-off that can be made between performance, resource usage and programmability.

The study of mapping multiple copies of the *MatVec* kernel showed that the number of *MatVec* IP blocks and the size of data (related to the number of cells to be processed) each block is processing varies from one approach to another. The final system design, whether auto-created by the system when using SDSoC OpenCL and SDSoC C++ or manually-created when using Vivado, has an effect on the number of *MatVec* IP blocks that can be created and on the maximum size of data each block can process. The SDSoC OpenCL design allowed the lowest number of *MatVec* IP blocks, a maximum of three *MatVec* IP blocks, each processing 42 cells, while the Vivado design allowed for the creation of twelve *MatVec* IP blocks each processing thirteen cells.

The exploration of kernel-to-kernel communication methods revealed that using the shared-memory method (DDR with SDSoC OpenCL and external BRAM with Vivado) between multiple kernels provided better performance compared with the use of pipes, the other explored data movement method explored. The use of DDR memory

in the SWM five kernels SDSoC OpenCL design and an external BRAM block in the Vivado SWM five kernels design resulted in the best performance as a kernel-to-kernel communication design option. In addition, in the multiple *MatVec* kernels design, the use of DDR memory in the SDSoC OpenCL and the SDSoC C++ solutions, and external BRAM blocks in the Vivado design was also a better design choice than the use of pipes. Further, the use of the shared-memory method was the most straightforward design solution in terms of programmability.

Chapter 8

Comparison Study (2): *MatVec* Kernel implementations in SDSoC OpenCL and SDSoC C++ Versus Vivado HLS

This chapter presents a comparison study of the implementation of the *MatVec* Vivado, SDSoC OpenCL and SDSoC C++ designs that we explored in Chapter 7. The comparative study is between the best performing *MatVec* designs from each approach with the use of the quantitative and qualitative metrics that we discussed in Section 3.4 from Chapter 3. A discussion of the results of each applied metric is presented and discussed in separate sections. The exploration study in Chapter 7 shows that the final *MatVec* designs we found for the SDSoC OpenCL and the SDSoC C++ approaches are different to the Vivado design in three main aspects. The first difference is that the Vivado design uses external BRAM blocks for providing data to the Vivado *MatVec* IP blocks, while DDR memory is used in the SDSoC OpenCL and C++ designs. This leads to the second difference which is that the SDSoC OpenCL and C++ kernel codes are slightly different from the Vivado kernel code due to the use of DDR memory. The third difference is the different used clock domains. The Vivado design runs on a higher clock frequency (310Mhz) than the SDSoC OpenCL (200 Mhz) and the SDSoC C++ (150Mhz). These differences are crucial facts in the comparison study in this chapter.

8.1 Performance Analysis

With the intention of having a fair like-for-like comparison of the *MatVec* designs, the initial aim was to replicate, as far as possible, the same design and coding decisions in the SDSoC OpenCL and C++ versions as were used in the pre-existing Vivado *MatVec* version. In this section we present the implementation results for each of the final *MatVec* design version that we explored in Chapter 7.

The final *MatVec* designs we found and explored in Chapter 7 have crucial differences that arise from two primary sources: first, from the different approaches to generate the overall system design and the use of FPGA resources, and secondly, from timing constraints in the designs which dictate the highest clock frequency at which a design can execute (the Xilinx Ultrascale+ board supports up to a 600 MHz clock). Moreover, each of the SDSoC OpenCL and C++ design methodologies impose other restrictions that have to be considered in the comparison.

In this section, we present the implementation performance figures of the *MatVec* designs described in Chapter 7 using two metrics. We are interested in the implementations performance in terms of the design's **Computational flop rate (Gflop/s)** and the **overall execution time** of the entire mini-App application. We first give a summary performance in terms of the *flop rate* of each implementation of the *MatVec* design and we analyse the design differences, scalability and peak performance of each implementation. We then discuss the performance in the context of each of the three phases of the overall algorithm, which, together, make up the overall execution time. The phases are: *Load*, preparing and loading data required by the FPGA IP blocks from the host, *Comp*, the compute time in the kernels, and *store*, returning the data from the FPGA blocks to the host.

8.1.1 Computation Flop Rate (Gflop/s)

To calculate the *MatVec* design's *floating point* operation rate we measure the time of the execution for the *MatVec* kernel to calculate the results for the full 864 cells (*tcalc*) in the mini-App. Based on that time, we calculate the time that the kernel took to calculate only one cell (*tpc_calc*), using equation 8.1. As we described in Section 2.6.2 from Chapter 2, we know that one cell has two flop operations (one addition and one multiplication) in the inner loop, so the total flops to be calculated per cell is $40 \times 6 \times 8 \times 2 = 3840$ *flops_per_cell*. If the created *MatVec* kernel design is a multi-cell design, then we multiply the *flops_per_cell* by the number of cells

that the kernel processes in one call resulting in a figure for `flops_per_OneCall`, as in equation 8.2. To calculate the total number of flops that a *MatVec* kernel is processing per second we divide the `flops_per_cell` by the kernel's `tpc_calc`, with equation 8.3. The `total_flop_rate` computed in equation 8.3 produces the *MatVec* design performance in flop/s but we convert it to Gflop/s for presentation.

$$tpc_calc = \frac{tcalc}{864} \quad (8.1)$$

$$flops_per_OneCall = flops_per_cell * number_of_cells \quad (8.2)$$

$$total_flops_ = \frac{flops_per_cell}{tpc_calc} \quad (8.3)$$

MatVec Vivado Design

Figure 8.1 shows the performance (Gflop/s) of the Vivado design implementation for a range of number of IP blocks and cells-per block (i.e. the data for the number of cells in the external BRAM of an IP block). As discussed in Section 7.2 from Chapter 7, the

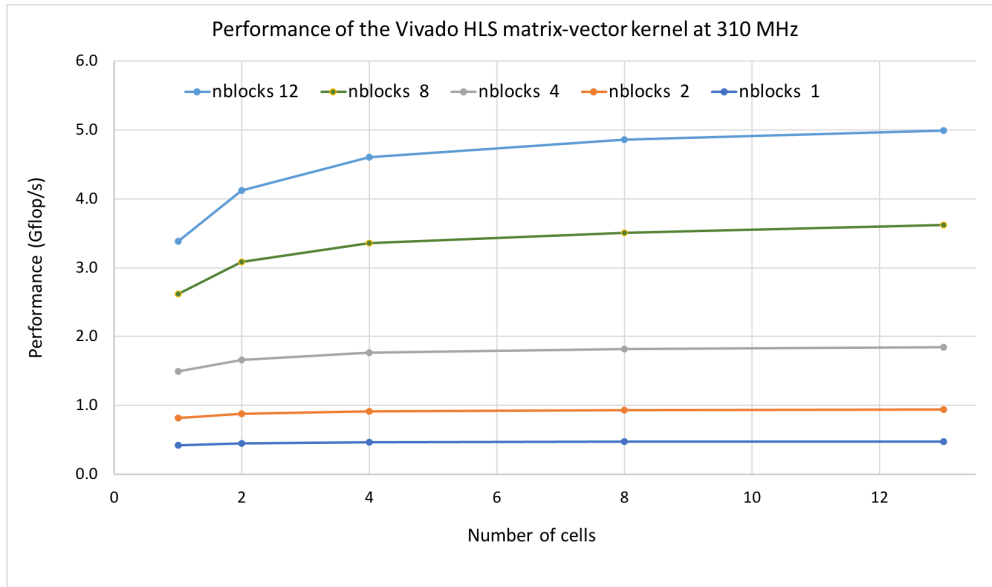


Figure 8.1: Performance of the Vivado Matrix-vector kernel designs at 310Mhz, as the number of blocks and cells-per-block varies. Performance figures here are the kernel "compute time only"

Vivado design has the advantage of providing BRAM memory that is created outside

the kernel IP blocks. In this design, the data is directly transferred from the host CPU DDR memory to BRAM associated with each block. BRAM access is much faster than access to DDR. In addition, as stated in [ARAM19], in any full port of the LFRic mini-app, the aim would be that the kernel data are generated on the FPGA and kept in the “FPGA plane” and so will not need to be repeatedly transferred to or from the host. A single *Matvec* IP block can run at 435 MHz, but when integrating 12 blocks (Max number of blocks) in a multi-block design, the maximum clock speed is reduced to 310 MHz to meet timing constraints. All data in Figure 8.1 is at 310 MHz.

The best performing Vivado implementation has 12 blocks and 13 cells-per-block and delivers 4.98 Gflop/s, as can be seen in Figure 8.1. The scaling with number of blocks is good, the parallel efficiency is 88%, i.e. the performance at twelve blocks as a fraction of twelve times the single block performance. As can be seen in Figure 8.1, increasing the number of cells per block initially delivers improved performance, as additional cells hide some of the latency costs, but quickly saturates.

Analysis of the design timeline reveals that for one block, there are 2 flops per cycle (one *dmul* and one *dadd*). At 310 Mhz and for 12 blocks, this leads to a theoretical peak performance figure of 7.44 Gflop/s. Thus, the Vivado HLS design achieves 67% of peak ¹.

***MatVec* SDSoC OpenCL Design**

Figure 8.2 shows the performance (Gflop/s) for three SDSoC OpenCL *MatVec* IP blocks and a range of cells-per-block. Unlike the Vivado implementation, where the kernel data is pre-loaded into fast external BRAM blocks, the data in the OpenCL implementation is located in DDR RAM. The OpenCL kernel block has to first load cells data into local BRAM before starting execution. As discussed in Section 7.3 from Chapter 7, the SDSoC OpenCL kernel processes multiple cells in a call to minimise the DDR memory access, whereas Vivado processes only a single cell in each call since the used memory solution is external BRAM blocks. See also the data movement design analysis in Section 8.3 for more insight.

The highest clock rate we were able to use in the OpenCL design is 200 MHz since above 200 MHz, the hardware design that is automatically generated by the SDSoC system did not meet timing constraints. The highest performance achieved is with one

¹Theoretical peak performance is calculated using this simple calculation: Frequency * the number of flops per cycle * the number of IP blocks. Applying this to the *MatVec* Vivado design (310*2*12 = 7.44 GF/s)

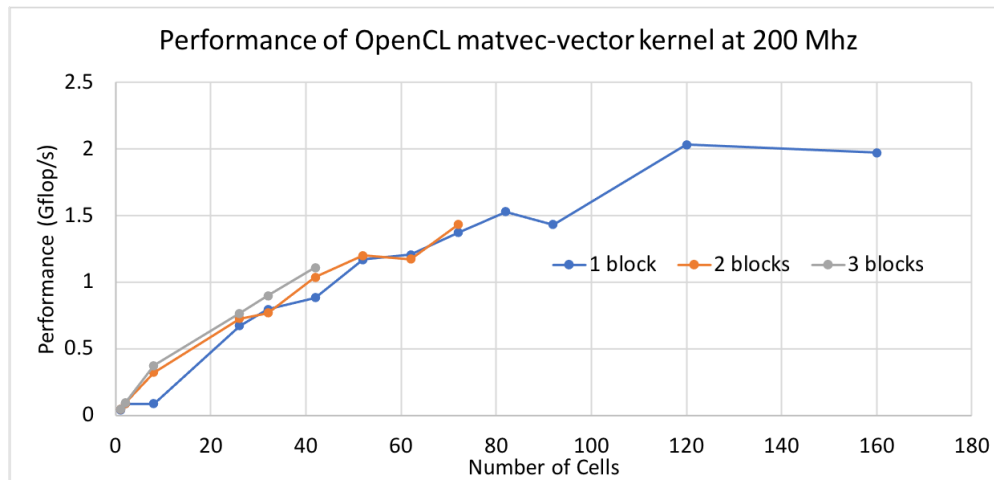


Figure 8.2: Performance of the OpenCL Matrix-vector kernel designs at 200Mhz as the number of blocks and cells-per-block varies. Performance figures here are the kernel "compute time only"

SDSoC OpneCL *MatVec* IP block with 120 cells, at 2.02 Gflop/s. SDSoC OpenCL implementations with different numbers of blocks scale quite well with increasing cell numbers, but the more blocks created, the fewer cells each block can have due to resource limitations, specifically BRAMs. The resources used for the best performing implementations are shown in Table 8.2 in Section 8.2 which shows the number and percentage used for each type of logic element per IP block and also for the total system design on the ZU9 FPGA. The highest number of SDSoC OpenCL *MatVec* IP blocks that could be created was only three, regardless of the number of cells, due to either lack of sufficient resources or failure to meet timing constraints. In addition, the creation of two or three SDSoC OpenCL *MatVec* IP blocks shows no performance benefit due to DRAM overhead. This is believed to be due to additional complexity in the auto-generated design due to the number of DDR memory paths required. This problem is avoided in the Vivado design, in which each block has a dedicated path to BRAM, rather than sharing limited paths to DDR.

The best Vivado design has 12 blocks with 13 cells in each block. The 12 blocks are executed concurrently, leading to the processing of 156 cells per 12 kernel execution. The best SDSoC OpenCL design has 1 block with 120 cells, a slightly lower number of cells. The OpenCL runs at 200 MHz while the Vivado runs at 310 MHz, which contributes to the performance differences seen. Analysis of the design timeline reveals that in the compute part of the kernel there are a maximum of 32 flops per cycle (16 dmuls and 16 dadds). At 200 MHz this leads to a theoretical peak performance figure of

6.4 Gflop/s (compared with 7.44 Gflop/s for Vivado HLS). Thus, the SDSoC OpenCL design achieves 31.5% of peak performance (Vivado achieves 67%).

MatVec SDSoC C++ Design

Figure 8.3 shows the performance (Gflop/s) of the SDSoC C++ design implementation as the number of blocks and cells varies. The highest performance with the SDSoC C++ is 1.78 Gflop/s with one block and 180 cells.

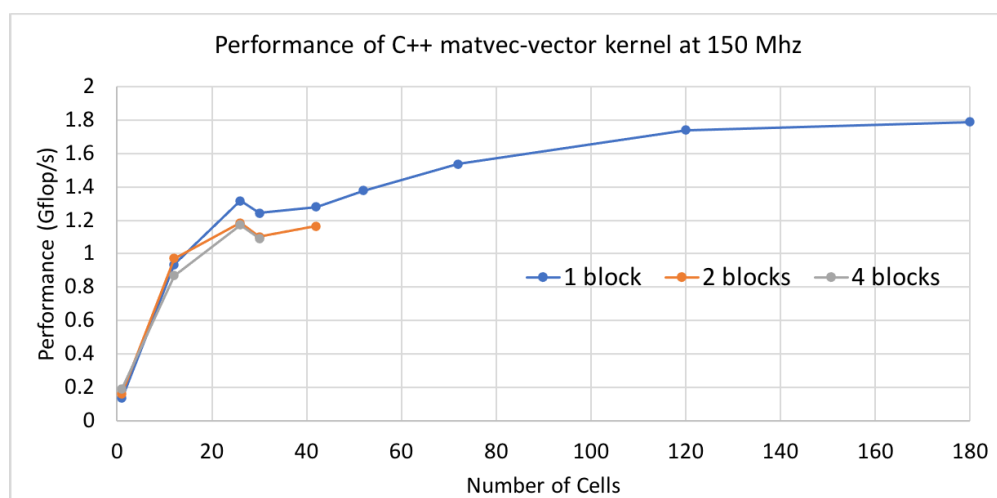


Figure 8.3: Performance of the C++ Matrix-vector kernel designs at 150Mhz, as the number of blocks and cells-per-block varies. Performance figures here are the kernel "compute time only"

As with the SDSoC OpenCL *MatVec* design, the SDSoC C++ performance figures include the data transfer costs from DDR to local BRAM. Another issue affecting the SDSoC C++ performance is the choice of DMA engine. The SDSoC C++ data engine that is used is the Scatter-Gather, SG, engine. SG is the slower DMA option, but it can handle larger volumes of data, making it the only option for transferring the large quantity of data associated with multiple cells. This issue is discussed further in the data movement design section below, Section 8.3.

The highest clock rate we were able to use in the SDSoC C++ design is 150 MHz, again limited by timing constraints in the auto-generated design. For the SDSoC C++ designs, the performance scales reasonably well with the increase in cell numbers, but the more blocks used, the fewer cells can fit in the FPGA due to resource limitations. Unlike SDSoC OpenCL, up to four blocks can be used with the SDSoC C++, as long

as the number of cells is small enough that resource limits are not exceeded. With more blocks, the design fails to build due to timing constraints.

In Figure 8.3, it can be seen that multiple block designs did not scale well as the number of blocks increases. This is as a result of the bandwidth of the DDR memory ports being shared between the blocks in the C++ design. In addition, the creation of two or four SDSoC C++ *MatVec* IP blocks shows no performance benefit due to DRAM overhead.

Analysis of the design timeline reveals that in the compute part of the kernel there are a maximum of 32 flops per cycle (16 dmuls and 16 dadds). At 150 MHz this leads to a theoretical peak performance figure of 4.80 Gflop/s (compared with 7.44 Gflop/s for Vivado HLS). Thus, the SDSoC C++ design achieves 37.5% of peak performance (Vivado achieves 67%).

8.1.2 Application Runtime Considerations

In this section, we compare the execution times of the three best *Matvec* designs in the three approaches in the context of three phases of the full application (Load, Comp and Store). This enables us to put the previous section's performance rates into the broader context of the whole application (overall execution time) and provides further insight into the different data movement and cells-per-block strategies that were employed in the designs of the three approaches. The overall execution time consists of the time to prepare the input data for use by the kernel(s) (Load time), the time that the kernel takes to do the arithmetic computation (Comp), and the time taken to store the results back in the data structures of the host (Store).

Note that the Load and Store times are not equivalent between the three approaches due to the different data movement methods used, as discussed in Section 8.3.

Table 8.1 shows the execution times for the best implementations and for two special cases (marked with an asterisk in the table) where the overall time is better.

For the SDSoC OpenCL and C++ designs we found that the highest *Comp* performance does not necessarily lead to the fastest overall time. For both SDSoC OpenCL and C++ we found that execution of 1 block with 26 cells delivered a better overall time compared that of the best performing versions - see the (*) data in Table 8.1. This is despite the fact that with 26 cells the *Comp* performance rate is only 0.67 Gflop/s (OpenCL) and 1.31 Gflop/s (C++) compared with 2.02 Gflop/s for the 120 cell SDSoC OpenCL version and 1.78 Gflop/s cells for SDSoC C++. The 26 cell case delivers faster overall time because the *Load* and *Store* times are faster. We note that for the 26

Table 8.1: Performance details for the best Matvec implementations from Vivado, SD-SoC OpenCL and SDSoC C++. (*: performance results for a smaller, 1 block/26 cell versions)

Design	Load Time(s)	Comp Time(s)	Store Time(s)	Overall Time(s)	Compute Time Performance (Gflop/s)	Overall Time Performance (Mflop/s)
Vivado 12-Blocks 13-Cells	0.02044	0.00067	0.01374	0.0349	4.98	95.06
SDSoC OpenCL 1-Block 120-Cells	0.02310	0.00489	0.01883	0.0469	2.02	70.74
*SDSoC OpenCL 1-Block 26-Cells	0.01867	0.00495	0.01951	0.0381	0.67	87.08
SDSoC C++ 1-Block 180-Cells	0.02234	0.00186	0.00327	0.0274	1.78	121.09
*SDSoC C++ 1-Block 26-Cells	0.01254	0.00251	0.00327	0.0183	1.31	181.29

cell case, the kernel is called more times than in the 120 cell OpenCL version and 180 cell C++ version, because it deals with fewer cells per call. This effect is not seen with the Vivado design.

With SDSoC C++, the *Load* and *Store* times are fastest with 1 block/26 Cells since, for the C++ design, data for 26 cells at a time is prepared by the host and written to DDR for the block, which has shared access with the FPGA.

Load time in the three best implementations is fairly similar, but the *Store* time is faster in the C++ implementations.

Vivado HLS *Load* and *Store* time is faster than the OpenCL time. For Vivado, data for 13 cells for each of 12 blocks (per kernel call) is prepared on the host at a time (in DDR memory) and then written directly from the host to the BRAM associated with each block. In contrast, for SDSoC OpenCL, data for 120 cells at a time is prepared on the host for the single block, and the OpenCL buffer write call executed (`enqueueWrite()`) [Khr21].

The Vivado *Comp* time is faster than that of both SDSoC OpenCL and C++. In SDSoC OpenCL and C++ the compute kernels are reading data from DDR and compute multiple cells per call, whereas in the Vivado design the compute kernels operate with data from BRAM and process one cell per call. However, the overall execution times show that the SDSoC OpenCL whole application performance is competitive with the Vivado design, while the SDSoC C++ whole application performance is faster in both the best implementation case and the 1 Block/26 cell case; mainly as a result of the cheaper *Store* times.

The results in the previous two Sections show that the Vivado design achieved the best performance for the compute phase with a performance peak around 4.98 Gflop/s, while the SDSoC C++ design achieved the best overall time. Although the SDSoC OpenCL and C++ designs operate at a lower clock rate than the Vivado design, and the data transfer time is from DDR rather than BRAM, the extra parallelism they exploit by processing multiple cells per call means that the SDSoC OpenCL design provides a competitive overall runtime, with a peak compute performance of 2.02 Gflop/s, and the SDSoC C++ design delivered the best overall time though with a peak compute performance of only 1.78 GFlop/s.

Table 8.2: The best implementations resource usage figures for the *MatVec* IP block and the total system design.

Resource Usage	Design	FF	LUT	DSP	BRAM
Usage Per IP Block	Vivado 12-Blocks 13-Cells	278388 (50.78%)	81396 (29.69%)	120 (5%)	48 (5.26%)
	OpenCL 1-Block 120-Cells	74845 (13%)	31174 (11%)	224 (8%)	588 (64%)
	C++ 1-Block 180-Cells	22895 (4%)	27563 (10%)	28 (1%)	781.5 (85%)
Total Design	Vivado 12-Blocks 13-Cells	304118 (55%)	178574 (65%)	120 (5%)	816 (90%)
	OpenCL 1-Block 120-Cells	94101 (17.16%)	48880 (17.83%)	224 (8%)	622.5 (68.25%)
	C++ 1-Block 180-Cells	49987 (9%)	37100 (14%)	28 (1%)	827 (91%)

8.2 Resource Usage Analysis

Table 8.2 presents the resource usage figures for the *MatVec* IP block and the total system design from the best Vivado, SDSoC OpenCL and SDSoC C++ implementations. Usage per IP block shows that the Vivado 12-blocks, 13-Cells design consumed more FF (50.78%) and LUT (29.69%) resources than the SDSoC OpenCL and the SDSoC C++ designs FF and LUT usage. The SDSoC C++ 1-Block, 180-Cells design consumed the lowest amount of FF (4%) and DSPs (1%), but it consumed the highest amount of BRAM resources (85%).

In terms of total design resource usage, the Vivado design utilisation of FF (55.78%) and LUT (65%) still higher than the SDSoC OpenCL and the SDSoC C++ designs FF and LUT usage. However, the utilisation of BRAM usage has increased to (90%), which is similar to the SDSoC BRAM usage of (91%).

8.3 Data Movement Analysis

This Section discusses the utilised methods for data transfer between the CPU and the memory solution (DDR RAM memory and the external BRAM blocks) in each approach, and the impact of these design choices upon performance.

Data movement in the Vivado design depends on isolating the data transfer from the kernel execution. The technique is achieved by providing external BRAM blocks for each kernel block, and the data transfer occurs in a separate step to the kernel call. The *matrix* and *x* data is read by the kernel from external BRAM into local BRAM, one cell at a time. The performance rates in Figure 8.1 exclude the data transfer time between CPU and FPGA. Table 8.3 shows that the Vivado load rate (1085.8 MB/s) is higher than the load rates in the best SDSoC OpenCL and C++ implementations. This is a result of the manually configured DDR-to-BRAM connection used in the Vivado design.

Table 8.3: Rates of data movement for the best implementations

Design	Load (MB/s)	Store (MB/s)
Vivado 12-Blocks 13-Cells	1085.8	108.2
OpenCL 1-Block 120-Cells	643.6	116.6
C++ 1-Block 180-Cells	666.9	676.7
*C++ 1-Block 26-Cells	1189.6	673.9

The load rates for the best SDSoC OpenCL (120 cells) and C++ (180 cells) implementations in Table 8.3 are slow compared to the Vivado load rate but, as discussed in Section 8.1.2, these rates reflect only the preparation of cell data and the writing of the data to DDR, rather than to the external BRAM blocks in the Vivado design. However, for the SDSoC C++ one Block/26 cells (which delivered the best overall time) load and store rates are better than those of the Vivado and the SDSoC OpenCL rates since the DDR buffers are shared between the host and device.

In SDSoC OpenCL and C++, local BRAMs are populated at the start of the kernel execution and this data is read during execution. In addition, the size of data transfer in the SDSoC OpenCL and C++ is larger than that of the data transferred between the external BRAM blocks and local BRAMs in the Vivado design. Thus, the data transfer overhead is counted in the SDSoC OpenCL and C++ performance. In addition, C++ design performance has been impacted by the DMA engine choices as described in Section 8.1.1.

8.3.1 Matvec CPU Implementation Comparison

The authors in [ARAM19] provided performance figures for the *MatVec* kernel implementation on a state-of-the-art Intel Broadwell E5-2650 v2 2.60 GHz CPU with eight cores. The CPU's eight cores are exploited by using OpenMP. Table 8.4 shows a performance comparison between the CPU implementation and the three best implementations we have discussed in this thesis. The CPU can deliver a peak performance

Table 8.4: Comparison of ZU9 FPGA double-precision Vivado, OpenCL and C++ matrix-vector performance implementations with Intel multicore CPU performance

Hardware	Performance (Gflop/s)	Peak performance (Gflop/s)	Percentage peak
Intel Broadwell E5-2650v2 2.60 GHz 8-core CPU	9.86	332.8	3.0%
ZCU102 FPGA (Vivado implementation)	4.98	600	0.83
ZCU102 FPGA (OpenCL implementation)	2.02	600	0.33
ZCU102 FPGA (C++ implementation)	1.78	600	0.29

of 332.8 Gflop/s as it can process 16 flops/cycle multiplied by eight cores with 2.6 GHz frequency. A theoretical ZCU102 FPGA peak Performance is 600 Gflop/s, as stated

in [Int21]². In [ARAM19] it was reported that FPGA performance for the double-precision matrix-vector kernel of is 5.34 Gflop/s which is 54% of that achieved on an 8- core Intel Broadwell CPU.

The CPU implementation outperforms the FPGA implementations. The ideal would be for the FPGA “accelerator” to outperform the CPU. However, considering the comparison between power consumption and price between these two devices is critical in an overall comparison with CPUs and other accelerators, such as GPUs. CPUs are much more power hungry than FPGAs. The authors in [Ber19] report that that GPU power efficiency is up to 20 Gflop/s/W, with price efficiency varying from 0.07 to 0.12 €/Gflop/s. GPUs are used as accelerators in multi-CPU systems as their efficiencies exceed those of CPUs. The author’s state that a mid-class FPGA’s power efficiency exceeds 70 Gflop/s/W, with a price efficiency of 0.29 €/Gflop/s.

8.4 Summary of the *MatVec* Comparison Study

In the case of the LFRiC matrix-vector-based mini-app studied, the overall performance gap between the three approaches is seen to be relatively small even though the original Vivado HLS solution could not be replicated exactly. While Vivado HLS provides a compute phase performance of 4.98 GFlop/s, SDSoc OpenCL 2.02 GFlop/s and SDSoc C++ 1.78 GFlop/s, the best overall execution times for one execution over the full mesh of 864 elements in this (small) version of the mini-app was 0.0183s (181.29 Mflop/s) for the SDSoc C++ with 1 Block/26 cells, whereas the overall time for the Vivado HLS design with 12 Blocks/13 cells was 0.0349s (95.06 Mflop/s) and the best SDSoc OpenCL overall time, with 26 cells, was 0.0381s (87.08 MFlop/s). The Vivado design compute performance benefited from not including the data transfer overhead. However, as the overall runtime is absolutely crucial here and probably what users are ultimately most interested in, the overall runtime of the SDSoc C++ design (1 Block/26 cells) make it the best design version.

The Vivado HLS design takes advantage of kernel-related BRAM, rather than DDR memory, to store input (and output) data prior to execution of the compute-intensive phase of the kernels; a clear performance advantage. This facility is not readily available in the SDSoc OpenCL and C++ approaches, and the consequent data access from DDR has a performance impact. In addition, the absence of this facility has limited

²The peak performance computation does not take into account the FPGA data precision; probably, it is an overestimate for 64-bit precision.

the scalability of multi-blocks performance in SDSoC OpenCL and C++ due to the sharing of DDR memory bandwidth between the blocks. However, the processing of multiple cells in a call allows the exploitation of some extra parallelism, providing a performance benefit over the Vivado HLS design which processes a single cell per call.

The abstraction level in the SDSoC OpenCL and the C++ approaches is higher than in Vivado HLS and leads to higher programmer productivity but provides less control of the system and low-level design than Vivado HLS, which can lead to extracting better performance from the FPGA resources.

8.5 Summary of the Two Comparison Studies

This section summarises the key findings from the two conducted comparative studies on the two benchmarks (SWM and MatVec).

8.5.1 Single kernel

As the exploratory study for the L100 single kernel shows, the Data-Flow mechanism with functions provided the best performance for mapping the L100 kernel's concurrency in both SDSoC OpenCL and Vivado approaches. The comparison study in chapter 6 showed that the L100 Data-Flow Vivado design achieved the best compute time (0.164s); however the L100 SDSoC OpenCL Data-Flow design provided the best overall application time (3.292s). Considering the data movement Load and Store costs, they were high in the single L100 kernel exploration, and dominated the overall time. The reason for this related to the fact that the data was shared with the host code in every call in a time-step (for 4000 time-steps) of the kernel. The cost of the data movement between the host and the L100 kernel with the use of the DDR memory in the SDSoC OpenCL implementations was lower compared to the use of external BRAM in the Vivado L100 design, as discussed.

8.5.2 Multiple kernels

The comparison study of the SWM five Kernels implementations in chapter 6 showed that the data movement cost was not significant, and did not affect the compute time of either the SDSoC OpenCL nor the Vivado design approach. The data in the five SWM kernels implementations were shared between only the five kernels through three data-movement solution options (DDR, Pipes, and external BRAM). The results revealed

that the Vivado SWM five kernels implementation provided the best compute time (1.334s) and overall time (1.337s). The external BRAM memory block shared between the five kernels in the Vivado design proved to be the best kernel-to-kernel communication design options. This is in contrast to the results seen with L100 Overall time where DDR was the better solution in terms of Load and Store time.

In the case of the LFRiC matrix-vector-based mini-app study, the Vivado design provided the best compute time performance (0.00067s) compared to the SDSoC OpenCL (0.00495s) and the SDSoC C++ (0.00251s). However, in terms of overall application time for one execution over the full mesh of 864 elements, the SDSoC C++ design with 1 Block/26 cells provided the best overall time of 0.0183s (181.29 Mflop/s); whereas the overall time for the Vivado HLS design with 12 Blocks/13 cells was 0.0349s (95.06 Mflop/s) and the best SDSoC OpenCL overall time, with 26 cells, was 0.0381s (87.08 MFlop/s).

As we have seen in the SWM and the *MatVec* studies, the Vivado design compute time performance benefited from not including the data transfer overhead. However, when we include the data transfer time, the SDSoC C++ in the *MatVec* study and the SDSoC OpenCL in the L100 study achieved better overall time than the Vivado design.

Comparing the SDSoC OpenCL, SDSoC C++, and the Vivado design methodologies, the Vivado design methodology with external BRAM solutions required higher development effort and hardware expertise. It requires the programmer to take care of many low-level concerns to create the FPGA solution, including the manual creation of the FPGA hardware system design, which requires low-level configuration, connections between the IP blocks, and address manipulation management. In managing the host and the kernel solutions, the Vivado approach also required more coding effort and FPGA expertise.

Chapter 9

Conclusions and Future Work

This chapter first summarizes the thesis findings, considering the extent to which the research questions raised in Chapter 1 have been addressed. This is followed by a review of the thesis contributions and a discussion of some limitations of the research which leads to a number of suggestions of possible research routes for future work.

9.1 Review of Thesis Research Questions

This thesis explored and compared the options and techniques for mapping the concurrency levels of two weather and climate applications using two high-level HLS tools, Xilinx SDSoC OpenCL and SDSoC C++, and a lower-level HLS tool, Xilinx Vivado to a single Xilinx Ultrascale+ FPGA board. Two exploratory and two comparison studies were conducted, involving a large number of experiments, in an attempt to answer the research questions (RQs) introduced in Chapter 1, and which are repeated in the following. In the exploratory studies, we collected data from a range of implementations and configurations of the two weather and climate applications, which we then utilized in the two comparative studies.

What are the technology options from the FPGA level and the HLS tool (SDSoC OpenCL and Vivado HLS) level for mapping the concurrency within a single HPC kernel and in the case of multiple HPC kernels?

We first explored the technology options from the FPGA level and the HLS tool level (focusing initially on SDSoC OpenCL and Vivado HLS) to map the concurrency within a single HPC kernel. The L100 kernel from the SWM application was chosen as the candidate kernel for this study. We first studied the concurrency types available in the algorithm for the L100 kernel and the possible mapping options available to

map these types of concurrency. Three (common) concurrency types were identified in the L100 algorithm (Instructions level parallelism, Data Parallelism, and Function parallelism). Using the SDSoC OpenCL and Vivado HLS approaches, twenty two FPGA designs were created to explore the mapping options.

The first exploratory study showed that targeting the mapping of the kernel's function-based parallelism using the mapping mechanism option `Data-Flow over functions` provided the best mapping design out of the twenty two FPGA designs of both approaches, SDSoC OpenCL and Vivado. In both approaches, the use of the `Data-Flow over functions` mechanism to map the kernel functional parallelism provided better execution time compared to the other mapping mechanisms that targeted the other concurrency types such as `Pipeline`, `Multiple Kernels`, `Unrolling` and `Dataflow over loops`. Processing the kernel's operations in parallel (coded as functions), where possible, was found to be the appropriate method to achieve good computational performance, if the kernel's operations are independent, as is the case in the operations in the L100 algorithm. In the the `Data-Flow over functions` L100 design implementation, the L100 kernel's independent operations (CU, CV, Z, H) are formed in the kernel code as four functions. These functions are then assigned to their own compute unit (a compute unit per function) in the generated FPGA design, and they are launched in parallel when the design is called from the host.

However, the `Data-Flow over functions` design style required higher programming effort than other options and resulted in higher resource consumption, specifically BRAM usage, than the other design options, illustrating the kind of trade-offs involved when searching the design space.

In contrast to the `Data-Flow over functions` design option, the design choice of using the `Pipeline` mechanism for mapping the L100 kernel's Instruction Level Parallelism (ILP) showed a competitive execution time and thus represents a viable alternative design choice. In this design choice, the L100 kernel's operations (CU, CV, Z, H) are coded using a single loop nest, where the pipeline pragma is applied over the inner loop of the kernel. Pipelining the inner loop was found to be the best method to extract the highest performance using the `Pipeline` mechanism. This design option required lower use of BRAM blocks compared to the `Data-Flow over functions` designs, providing a trade-off in design choices in a case where a design may require an extensive use of BRAM blocks, as we discuss next.

Two memory solutions (DDR and on-chip BRAM¹) were available for data sharing

¹on-chip BRAM is the BRAM block on the Programmable Logic Part of the MPSoC

between the host CPU and the FPGA solution for the exploratory studies. We utilized the DDR memory solution in the SDSoC OpenCL approach since, in OpenCL, there was no support for the direct implementation of on-chip BRAM shared between the host and FPGA. As we wished to explore both memory solutions, we utilized the on-chip BRAM memory solution (which we called *external BRAM*) in the Vivado designs which did support its use. This is a clear illustration of the point that the ability to manually manipulate the lower-level Vivado design allows more freedom in design choices.

The use of `Data-Flow over functions` with external BRAM in Vivado provides a trade-off between the number of external BRAMs and the bandwidth available. The best performing Vivado L100 `Data-Flow` FPGA design required the creation of seven external BRAMs, one for each data array, to achieve the best execution time. However, this exhausted the available BRAM, which has an implication on the multiple Vivado kernels design.

Following this single kernel exploration, the lessons learned for exploring the mapping of the L100 kernel concurrency types were used in an exploratory study of the technology options for mapping an application with multiple HPC kernels to FPGA. The SWM application with nine kernels was the case study for this, and we used the SDSoC OpenCL and Vivado approaches again for the implementation designs. We first studied the options for optimizing every single kernel based on the results we found in mapping the L100 kernel. In the SDSoC OpenCL approach, we mapped the SWM kernels that share similar concurrency types to the L100 kernel, such as L200 and L300, using the `Data-Flow over functions` mechanism. The other kernels that are not suitable for the `Data-Flow` coding style (functions), such as L100pc and L200pc, were mapped using the second-best mechanism option found, that is the `pipeline` mechanism (pipeline the inner loop).

However, in the Vivado approach, we mapped all the SWM kernels using the `pipeline` mechanism. The reason for this design decision related to the Vivado memory solution choice we made (external BRAM block). As we explored in the Vivado L100 mapping design, we initially tried creating seven BRAM blocks with the use of `Data-Flow over functions` mechanism aiming to achieve the best performance. However, applying the `Data-Flow over functions` design to the other kernels (L200 and L300) led to the requirement for many BRAM blocks to be created, which conflicted with the number of available BRAM resources on the Ultrascale+

FPGA. In addition, the Vivado design's complexity (creating many BRAM blocks, requiring more addresses configuration, and management of data distribution and movement) was much more labour intensive (and error prone). The trade-off of utilizing high BRAM resources (seven BRAM blocks) when implementing one kernel (L100) was acceptable to achieve higher performance, however, when implementing multiple kernels, this performance trade-off was not suitable. Consequently, we chose to map all the SWM kernels using the pipeline mechanism in the Vivado multiple kernels implementation.

Next, we explored the trade-offs involved in the number of kernels that can be implemented, based on the available resources in the FPGA board, in the context of the constraint on the largest possible problem size a given number of kernels can process. We found that five out of the nine SWM kernels, which implemented the computation in the main time-stepping loop (i.e. excluding kernels that were essentially implementing one-time initialization functions) was the appropriate number of kernels to place on the FPGA. The maximum problem size that these five kernels can process was found to be a 53×53 grid before exceeding the FPGA resources limit.

The final part of the multiple HPC kernels mapping exploration was exploring the technology options for managing the kernel-to-kernel data movement. Three design options were explored which were: DDR memory, external BRAM and pipes. The DDR memory and pipes were explored using the SDSoC approach, while the external BRAM was available to be explored only in the Vivado approach. The results showed that using the shared-memory method (DDR or external BRAM) between multiple kernels provided better performance against the other explored data movement method with pipes. This design decision was the best in both the SDSoC OpenCL and the Vivado kernel-to-kernel communication exploration studies. In addition, it was the most straightforward design solution in terms of programmability.

Can the use of high-level optimization techniques in SDSOC OpenCL and SDSoC C++ match the design choices and performance achievable from the use of the lower-level manual optimizations in Vivado HLS?

In the second exploratory study we undertook the mapping of an existing lower-level FPGA design, of the *MatVec* kernel from the LRFic application implemented using the Vivado approach, to two relatively higher-level HLS approaches, SDSoC OpenCL and SDSoC C++. We first explored the different mapping methods and optimisation techniques utilised in the Vivado *MatVec* design. We classified the Vivado *MatVec* design mapping techniques into three categories: the kernel source code, the

MatVec IP block and the system design in which it is embedded, and the host code design. This classification was used to try to replicate the Vivado *MatVec* design objectives where possible. This exploration study shows that some Vivado *MatVec* design aspects could be mapped successfully in the SDSoC OpenCL and the SDSoC C++ *MatVec* designs. However, other design choices did not, or could not, be applied to the SDSoC OpenCL and the SDSoC C++ *MatVec* designs. The manual design manipulation required in the Vivado approach allows for more design freedom and options compared to the SDSoC OpenCL and SDSoC C++ approaches. Two main Vivado design methods were not able to be replicated in the SDSoC OpenCL and C++ designs. The creation and use of external BRAM blocks outside the *MatVec* IP blocks and the ability to have two clock domains being two of the most significant. The SDSoC OpenCL and SDSoC C++ designs, therefore, used DDR memory and only one clock domain. The use of DDR memory influenced how the SDSoC OpenCL and SDSoC C++ kernels are designed. We had to make some necessary kernel design changes that are different to the Vivado *MatVec* kernel code in order to try to compensate for the use of DDR memory rather than the (lower latency) BRAM. Those changes were that the SDSoC OpenCL and C++ *MatVec* IP blocks process multiple data cells in a single kernel call, and the `matrix` data is transferred into the BRAM internal to the kernel IP block in full, rather than by slice as with Vivado. Therefore, the Vivado *MatVec* kernel code is different to the SDSoC OpenCL and the SDSoC C++ kernel codes. The Vivado *MatVec* IP block can run with a 310 MHz clock frequency because the Vivado *MatVec* system design has two separated clock domains. In contrast, the SDSoC OpenCL design could run at only 200 MHz, and the SDSoC C++ design at 150 MHz, which were found to be the maximum possible.

What can be said about the best mapping technology options suitable for HPC application's concurrency, and about the trade-offs related to achieving the best choice in terms of performance, resource usage, and development effort?

We conducted the first comparison study to gain insight about the best mapping technology options, in terms of performance, resource usage and development effort, in mapping the single and multiple SWM kernels using the SDSoC OpenCL and Vivado approaches. The comparison study analysed the performance results, resource usage figures, and design decisions for the L100 kernel concurrency mapping study and for the SWM five kernels exploration in both the SDSoC OpenCL and the Vivado approaches. The comparison results revealed that the Data-Flow mechanism with functions provided the best performance for mapping the L100 kernel's concurrency

in both SDSoC OpenCL and Vivado approaches. However, this mechanism required high BRAM resource usage and coding effort. The L100 Data-Flow Vivado design achieved the best compute time and the L100 SDSoC OpenCL Data-Flow design provided the best overall application time. The data movement (Load and Store) times were high compared to the compute time in the both SDSoC OpenCL and Vivado L100 implementations, and this dominated the overall time. However, the use of the DDR memory in the SDSoC OpenCL implementations provided better data movement (Load, Store) times compared to the Vivado L100 (Load, Store) multi-kernel times.

Nevertheless, in the SWM five Kernels implementations, the data movement cost was less significant, not affecting either approach's design compute time, because the data were only shared between the five kernels through one of the three data-movement solution options (DDR, Pipes, and external BRAM). The external BRAM memory block shared between the five kernels in the Vivado design provided the best compute and overall time in the kernel-to-kernel communication comparison.

What are the trade-offs between performance and programmer effort (which can be expected to be reduced) that can be achieved by using the high-level approaches of SDSoC OpenCL and SDSoC C++ compared to using the lower abstraction level of the Vivado HLS?

We also conducted a second comparison study to consider the trade-offs between performance and programmer effort (which can be expected to be reduced) that can be achieved by using the high-level approaches of SDSoC OpenCL and SDSoC C++ compared to using the lower abstraction level of the Vivado HLS. In doing so, we used the data collected from the second exploratory study for the *MatVec* kernel. This comparison study showed that the overall performance gap between the SDSoC OpenCL/C++ and Vivado approaches was seen to be relatively small even though the original Vivado HLS solution could not be replicated exactly. While Vivado HLS provides the best compute time performance, the best overall execution times for one execution over the full mesh of 864 elements in this (small) version of the mini-app was delivered by the SDSoC C++ with 1 Block/26 cells design. The Vivado design's compute performance was seen to have benefited from not including the data transfer overhead. However, when we include the data transfer time, the SDSoC C++ overall time was better than the Vivado design, and the SDSoC OpenCL overall time is competitive.

The Vivado HLS design takes advantage of kernel-related BRAM, rather than DDR memory, to store input (and output) data prior to execution of the compute-intensive phase of the kernels; a clear performance advantage. This facility is not available in the

SDSoC OpenCL and C++ approaches, and the consequent data access from DDR has a performance impact. In addition, the absence of this facility has limited the scalability of the multi-block performance in SDSoC OpenCL and C++ due to the sharing of DDR memory bandwidth between the blocks. However, the processing of multiple cells in a call allows the exploitation of some extra parallelism, providing a compute performance benefit over the Vivado HLS design which processes only a single cell per call.

In terms of programmability, comparing the three utilised approaches in this thesis (SDSoC OpenCL, SDSoC C++ and Vivado), the conducted studies show that the Vivado approach requires much knowledge that the traditional high-level programmer would not typically possess. It requires the highest amount of low-level hardware knowledge and much design experience to work through and evaluate the many implementation options and design trade-offs that are available, and this results in high development effort. This is because the approach requires the programmer to take care of many lower-level concerns to produce an appropriate, efficient FPGA solution. This includes the manual creation of the FPGA hardware system design, which requires the explicit configuration of the data-movement mechanisms and of the connections between the IP blocks, address manipulation, setting widths of data paths and managing the execution of the kernel IP blocks through setting and examining the start and stop bits of the kernel. In addition, the programmer has to design and implement the management of the preparation and transfer of data to the FPGA BRAM blocks explicitly in the host code. This all requires significant effort. The writing and optimization of kernel code requires a similar level of effort to other methods, however; essentially, only the syntax of the pragmas required for dataflow, pipelining and unrolling, etc. change. In contrast, the OpenCL and C++ approaches hide most of the complexity of the system design steps required by the Vivado approach and assign it to the SDSoC compilers. In these two approaches, the programmer needs only to know the SDSoC design flow and to have a good understanding of the kernel code pragmas for dataflow, pipelining and unrolling, etc., and knowledge of their expected effect on the performance. The C++ approach requires also an understanding of the data-movement engines available in the system and their advantages and disadvantages. However, the resulting automatic designs, while generally more simple, are not as efficient as can be achieved manually with Vivado. This is presumably because of the generality of the design solution, but this generality can lead to issues with timing in designs, limiting the maximum clock rate that can be used, for example.

In conclusion, as with all sophisticated systems, the more low-level knowledge a programmer/designer has, the better performance that can generally be achieved. The current level of the abstraction in the higher-level HLS tools such as SDSoC OpenCL and SDSoC C++ cannot necessarily help programmers to achieve better performance. However, in the research in this thesis we have seen that a competitive performance and, in the case of the *MatVec* kernel, a better overall application time (including Load, compute and Store time) can be achieved with less effort than with the Vivado approach.

Is it feasible to consolidate a methodology for mapping the concurrency in weather and climate applications to FPGAs to improve the FPGA programmability for traditional HPC software programmers?

The **design space exploration** undertaken in this thesis leads us to conclude that consolidating such a methodology would require extensive gathering of data from both across HPC weather and climate applications and the many HLS tools that are emerging in order to reach an acceptable level of generalization to enable a general methodology to emerge.

This thesis studied, in detail, only two HPC weather and climate applications cases using only one set of tools from one vendor, Xilinx (SDSoC OpenCL, SDSoC C++, Vivado), targeting only a single FPGA (an Ultrascale+ SOC Architecture board). In the available time frame for this thesis, we aimed for an exhaustive exploration as was possible for exploring options and mechanisms that the tools provided, with the intention of exploring the (vast) design space and this required long hours as a result of the fact that producing FPGA designs, building implementations, generating bitstreams and debugging etc. processes are slow.

Thus the development/debug cycle for each design option case in this thesis required a long time, which limited the amount of data that could be gathered during the time available which would contribute towards the aim of consolidating a methodology. There is now a growing body of literature that is beginning to provide other data that might be useful in a future step to consolidate the lessons being learned across the published studies. However, many of these studies tend to present a "best" design, without showing the path (and trade-offs) explored to achieve that design, which limits their use when attempting to generalize.

Despite that this thesis did not fully achieve this target, it does contribute by providing data from a wide exploration of two cases studies using three HLS tools with a considerable variation of FPGA designs which can hopefully feed into work towards

a future methodology consolidation. In addition, many lessons can be learnt from this research, as the SWM benchmark is a stencil-based algorithm, a common pattern in HPC applications; And the research findings can also be applied to other HLS tools such as Vitis.

9.2 Summary of Contributions

The contributions in this thesis have resulted from a thorough **design space exploration** of the utilization of one set of Xilinx HLS tools (SDSoC OpenCL, SDSoC C++, Vivado) to exploit concurrency types in two weather and climate-related applications targeting an SOC Xilinx Ultrascale+ FPGA. The following points summarise thesis contributions.

An exploratory study of the use of HLS mechanisms and options available from a high-level HLS technology approach, SDSoC OpenCL, and a lower-level HLS approach, Vivado, for mapping the concurrency in a single candidate HPC kernel from the Shallow Water *SWM* application to a Xilinx Ultrascale+ FPGA. Twenty two FPGA designs variation were created for conducting the exploration of mapping instruction-level-, data- and functional-parallelism. Variants of host CPU techniques from the OpenCL language and kernel code techniques such as `Dataflow` and `Multiple kernels` were utilized for conducting this exploration. Moreover, four five-kernel SWM FPGA designs were created to explore the technology options for managing multiple SWM kernels and kernel-to-kernel communications mechanisms, such as DDR memory, on-chip memory and pipes, to map the whole HPC application to a Xilinx Ultrascale+ FPGA.

An exploratory study for investigating the extent to which it is possible to achieve replication of an existing FPGA design for the *MatVec* kernel, created using the (lower-level) HLS tool Vivado, in the (higher-level) HLS tools SDSoC OpenCL and SDSoC C++. Eleven FPGA design variations were created to explore similarities, differences, and options for design decisions, including optimization levels and mapping methods for the host CPU codes, the kernel codes, and the FPGA hardware system designs in the three HLS approach. Various optimization techniques and methods in the host, kernel code, and at the hardware-level were explored. Three *MatVec* hardware designs were presented and discussed at a low-level of abstraction.

A comparison study and analysis was presented of the performance (execution

time and overall application time), resource usage, and programmability issues between the SDSoC OpenCL and Vivado HLS for mapping the candidate SWM single kernel L100 to a Xilinx Ultrascale+ FPGA. In addition, a comparison study was presented of the performance, resource usage and programmability of the different possible solutions of mapping the SWM multi-kernel and methods for managing the kernel-to-kernel communication. We compared and presented a detailed analysis of the best performing design variations out of the twenty two L100 designs created and the four five-kernel SWM FPGA designs.

A **comparison study and analysis** was presented of Vivado HLS and SDSoC OpenCL and SDSoC C++ *MatVec* implementations along with an analysis of the techniques available to be used in the three approaches with a focus on the differences in the three approaches which result in their performance, scalability and resource usage, along with a discussion of programmability issues. We compared and discussed the source of inefficiencies and the design limitations in the three best performing *MatVec* FPGA designs. We also briefly compared the FPGA design's performance against a multi-core CPU *MatVec* design.

9.3 General Recommendations

These recommendations and insights are concluded from the learned lessons in this thesis. In porting HPC kernels to FPGAs, we recommend identifying first the concurrency types in the kernel. In addition, studying the possible best options for mapping the identified concurrency types and finding the best trade-off mapping mechanism option. In selecting the HLS tool, it is recommended to understand the trade-off in how deep hardware knowledge is needed to utilize that HLS tool, which would affect productivity and performance gaining. It is important to consider studying the trade-off between the kernel's problem size and the level of optimization that can be applied, as this trade-off relates to the available FPGA resources. If the FPGA resources trade-off allows for creating multiple IP blocks, it is recommended to consider applying spatial parallelism over duplicated IP blocks. It is recommended to utilize the dataflow style within the IP block to explore more parallelism if the kernel's instructions are independent. In debugging design bugs, (first) testing the kernel's output is recommended. Dataflow and pipes coding styles are the most prone to error. Therefore, applying these two coding styles with careful data dependency and memory bandwidth management is recommended. In the Vivado HLS approach, it is recommended to carefully manage

the data port width and data/IP block addresses, as they cause the most errors.

9.4 Limitations and Future Work

This thesis explored possible solutions and options for mapping two weather and climate HPC application to FPGAs which are available to the HPC programmer from two high-level HLS tools and one lower-level HLS tool.

The concurrency mapping designs we have explored in the two exploratory studies have performance limitations due to the cost of data movement. Although we applied optimization strategies to improve the data movement, the results show that a more in-depth study is needed in the future. The following data movement optimization strategies can be the subject of future work.

Utilise On-chip global memory: BRAMs can be used as OpenCL global memory. The use of global memory shared between kernels would reduce the overhead of DDR access. In addition, on-chip global memory would allow for future study of the performance of OpenCL NDRange kernels. We have seen that using on-chip global memory in the Vivado designs has performance benefits, but its use currently requires high programming effort. On-chip global memory is not supported in the SDSoC HLS tool 2018.2 used in this work or in newer versions up to 2019 (though there was an attempt to support it in earlier versions).

Utilising the full width of DDR ports: The second optimization strategy is utilizing the full width of the DDR ports. This designs in this thesis, using either SDSoC OpenCL/C++ or Vivado, transfer data with only the bit-width of the data type (double or float). Attempts to widen the data transfer width were left to the tools' compiler to decide. We have seen that in only one case (SDSoC OpenCL MatVec designs) the `xocc` compiler has chosen to apply 512 bit-width. This method would considerably reduce the data-movement overhead. The manual attempts to apply this strategy were unsuccessful. To apply such a method, we needed to use a bigger data type such as `double16`. However, this data type is not supported in the SDSoC HLS tool 2018.2.

Utilise DDR memory banks: The third strategy is the utilization of the DDR memory banks. For example, the DDR memory on the Zynq UltraScale+ MPSoC ZCU102 board has four memory banks; these banks can be accessed from the kernel in parallel. Utilizing this method would reduce the data-movement overhead and improve the performance of the implementation. This feature is supported in the new Xilinx tool, Vitis, but this tool appeared too late in the timescale of this thesis, and is, therefore, out

of the thesis scope and its use subject to future work.

Designing a solution in the Vivado approach involves having to consider selecting from a great deal of various design options and methods. In this thesis we decided to follow certain design choices that suited the study objectives and scope. There are other interesting design options, such as the use of data streaming (pipes), alternative data movement engines and other uses of DDR memory which are left to future investigations.

This thesis focused on exploring the higher level of options and mechanisms available in the selected HLS tool, with the perspective of a traditional HPC programmer; a future in-depth study of low-level FPGA hardware aspects could help improve and understand the outcome of the user-guided, but auto-compiled designs resulting from the tools.

Bibliography

- [AEB⁺16] Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Omar Hammami, and Ismail Ali. High-level synthesis for FPGA design based-SLAM application. In *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pages 1–8. IEEE, 2016.
- [AFH⁺19] Samantha V Adams, Rupert W Ford, M Hambley, JM Hobson, I Kavčič, Christopher M Maynard, Thomas Melvin, Eike Hermann Müller, S Mullerworth, AR Porter, et al. LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models. *Journal of Parallel and Distributed Computing*, 132:383–396, 2019.
- [Alg20] Moteb Alghamdi. Design space exploration of concurrency mapping to fpgas with opencl: A case study with shallow water model kernel. In *Proceedings of the International Workshop on OpenCL*, pages 1–1, 2020.
- [AMI21] Mikhail Asiatici, Damian Maiorano, and Paolo Ienne. How many CPU cores is an FPGA worth? Lessons learned from accelerating string sorting on a CPU-FPGA system. *Journal of Signal Processing Systems*, pages 1–13, 2021.
- [ARA21a] Moteb Alghamdi, Graham Riley, and Mike Ashworth. A Comparison of Vivado HLS, SDSoc C++ and OpenCL for Porting a Matrix-vector-based Climate model mini-app to FPGAs. In *PDPTA’21-The 27th Int’l Conference on Parallel and Distributed Processing Techniques and Applications*, 2021.
- [ARA21b] Moteb Alghamdi, Graham Riley, and Mike Ashworth. Concurrency Mapping to FPGAs with OpenCL: A Case Study with a Shallow Water Kernel. In *CSC’21-The 19th Int’l Conf on Scientific Computing*, 2021.

- [ARAM19] Mike Ashworth, Graham D Riley, Andrew Attwood, and John Mawer. First steps in porting the LFRic weather and climate model to the FPGAs of the EuroExa architecture. *Scientific Programming*, 2019.
- [Ash10] Peter J Ashenden. *The designer's guide to VHDL*, volume 3. Morgan Kaufmann, 2010.
- [BD19] Nick Brown and David Dolman. It's all about data movement: Optimising fpga data access to boost performance. In *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 1–10. IEEE, 2019.
- [Ber19] BertonDSP. GPU vs FPGA Performance Comparison, Berton White Paper BWP001 v1.0. http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf., 2019. [Online; accessed 12-January-2021].
- [BFV⁺17] Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. Exploiting parallelism on GPUs and FPGAs with OmpSs. In *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*, pages 1–5, 2017.
- [BKB21] Nick Brown, Mark Klaisoongnoen, and Oliver Thomson Brown. Optimisation of an fpga credit default swap engine by embracing dataflow techniques. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 775–778. IEEE, 2021.
- [BNM⁺20] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C. Hoe, Vaughn Betz, and Martin Langhammer. Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 10–19, 2020.
- [Bro19] Nick Brown. Exploring the acceleration of the Met Office NERC cloud model using FPGAs. In *International Conference on High Performance Computing*, pages 567–586. Springer, 2019.

- [Bro21] Nick Brown. Porting incompressible flow matrix assembly to fpgas for accelerating hpc engineering simulations. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 9–20. IEEE, 2021.
- [BRS13] David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses: The programmability of FPGAs must improve if they are to be part of mainstream computing. *Queue*, 11(2):40–52, 2013.
- [CBM⁺18] Georgios Christodoulis, François Broquedis, Olivier Muller, Manuel Selva, and Frédéric Desprez. An FPGA target for the StarPU heterogeneous runtime system. In *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8. IEEE, 2018.
- [CCS⁺18] Mattia Cacciotti, Vincent Camus, Jeremy Schlachter, Alessandro Pezzotta, and ChristianENZ. Hardware acceleration of HDR-image tone mapping on an FPGA-CPU platform through high-level synthesis. In *2018 31st IEEE International System-on-Chip Conference (SOCC)*, pages 158–162. IEEE, 2018.
- [CFH⁺18] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. Best-effort FPGA programming: A few steps can go a long way. *arXiv preprint arXiv:1807.01340*, 2018.
- [Cha12] Julius Chang. *General circulation models of the atmosphere*, volume 17. Elsevier, 2012.
- [CHPB21] Walther Carballo-Hernández, Maxime Pelcat, and François Berry. Why is FPGA-GPU Heterogeneity the Best Option for Embedded Deep Neural Networks? *arXiv preprint arXiv:2102.01343*, 2021.
- [CLN⁺11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Visers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.
- [DRP11] Rob Dimond, Sébastien Racaniere, and Oliver Pell. Accelerating large-scale HPC Applications using FPGAs. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 191–192. IEEE, 2011.

- [DVKG05] Yong Dou, Stamatis Vassiliadis, Georgi Krasimirov Kuzmanov, and Georgi Nedeltchev Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 86–95, 2005.
- [FOS⁺14] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2014.
- [GBLS16] Paul Grigoraş, Pavel Burovskiy, Wayne Luk, and Spencer Sherwin. Optimising sparse matrix vector multiplication for large scale fem problems on fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2016.
- [GCDJ19] Matthias Goebel, Kai Norman Clasen, Robert Drehmel, and Ben Jurlink. Evaluating the Memory Architecture of Next-Generation FPGA-SoCs for HPC. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 209–216. IEEE, 2019.
- [GF20] Atharva Gondhalekar and Wu-chun Feng. Exploring FPGA Optimizations in OpenCL for Breadth-First Search on Sparse Graph Datasets. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 133–137. IEEE, 2020.
- [GFY⁺14] Lin Gan, Haohuan Fu, Chao Yang, Wayne Luk, Wei Xue, Oskar Mencer, Xiaomeng Huang, and Guangwen Yang. A highly-efficient and green data flow engine for solving euler atmospheric equations. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, 2014.
- [GLN⁺14] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

- [GLR19] Georgios Georgis, George Lentaris, and Dionysios Reisis. Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution. *Journal of Real-Time Image Processing*, 16(4):1207–1234, 2019.
- [HLC⁺13] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis for fpgas. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 89–96. IEEE, 2013.
- [HM17] Nicole Hemsoth and Timothy Prickett Morgan. *FPGA Frontiers: New Applications in Reconfigurable Computing*. Next Platform Press, am, 2017.
- [HM21] Rania O Hassan and Hassan Mostafa. Implementation of deep neural networks on FPGA-CPU platform using Xilinx SDSOC. *Analog Integrated Circuits and Signal Processing*, 106(2):399–408, 2021.
- [HP19] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.
- [HSS88] G-R Hoffmann, PN Swarztrauber, and RA Sweet. Aspects of using multiprocessors for meteorological modelling. In *Multiprocessing in meteorological models*, pages 125–196. Springer, 1988.
- [Int] Intel HARP system. https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf. Accessed: 2019-11-12.
- [Int20] Intel. Intel FPGAs SDK for OpenCL. 2-<https://www.intel.co.uk/content/www/uk/en/software/programmable/sdk-for-opencl/overview.html>, 2020. [Online; accessed 01-September-2020].
- [Int21] Intel. Understanding peak floating-point performance claims. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf>, 2021. [Online; accessed 12-January-2021].

- [JF20] Zheming Jin and Hal Finkel. Population Count on Intel® CPU, GPU and FPGA. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 432–439, 2020.
- [JZ16] Qi Jia and Huiyang Zhou. Tuning stencil codes in OpenCL for FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 249–256. IEEE, 2016.
- [Ken19] Tobias Kenter. Invited Tutorial: OpenCL design flows for Intel and Xilinx FPGAs: Using common design patterns and dealing with vendor-specific differences. In *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*, pages 1–8. VDE, 2019.
- [KG17] Lester Kalms and Diana Göhringer. Exploration of opencl for fpgas using sdaccel and comparison to gpus and multicore cpus. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.
- [Khr21] Khronos. opencl-cplusplus Specs. <https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf>, 2021. [Online; accessed 12-January-2021].
- [KHZ16] Dirk Koch, Frank Hannig, and Daniel Ziener. *FPGAs for Software Programmers*. Springer, 2016.
- [KJPN10] Vinay BY Kumar, Siddharth Joshi, Sachin B Patkar, and H Narayanan. FPGA based high performance double-precision matrix multiplication. *International journal of parallel programming*, 38(3):322–338, 2010.
- [KPJ⁺21] Martin Karp, Artur Podobas, Niclas Jansson, Tobias Kenter, Christian Plessl, Philipp Schlatter, and Stefano Markidis. High-performance spectral element methods on field-programmable gate arrays: Implementation, evaluation, and future projection. In *35rd IEEE International Parallel & Distributed Processing Symposium*, 2021.
- [KPK⁺22] Martin Karp, Artur Podobas, Tobias Kenter, Niclas Jansson, Christian Plessl, Philipp Schlatter, and Stefano Markidis. A high-fidelity flow solver for unstructured meshes on field-programmable gate arrays: Design, evaluation, and future challenges. In *International Conference on*

- High Performance Computing in Asia-Pacific Region*, pages 125–136, 2022.
- [KPZ⁺16] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 115–127. Ieee, 2016.
- [KSFNA21] Tobias Kenter, Adesh Shambhu, Sara Faghih-Naini, and Vadym Aizinger. Algorithm-hardware co-design of a discontinuous galerkin shallow-water model for a dataflow architecture on fpga. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–11, 2021.
- [LWY⁺17] Yingyi Luo, Xianshan Wen, Kazutomo Yoshii, Seda Ogrenci-Memik, Gokhan Memik, Hal Finkel, and Franck Cappello. Evaluating irregular memory access on opencl fpga platforms: A case study with xsbench. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.
- [LY16] Huan Li and Wenhua Ye. Efficient implementation of FPGA based on Vivado high level synthesis. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 2810–2813. IEEE, 2016.
- [Max20] Maxeler. Maxeler SDK. <https://www.maxeler.com/products/desktop/>, 2020. [Online; accessed 01-September-2020].
- [MGMG11] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.
- [MML] ALEKSANDAR MILINKOVIĆ, STEVAN MILINKOVIĆ, and LJUBOMIR LAZIĆ. FPGA based dataflow accelerator for large matrix multiplication.
- [MPK21] Johannes Menzel, Christian Plessl, and Tobias Kenter. The strong scaling advantage of fpgas in hpc for n-body simulations. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(1):1–30, 2021.

- [MV14] Sparsh Mittal and Jeffrey S Vetter. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):1–23, 2014.
- [MVBG⁺12] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today’s high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.
- [MWT⁺20] Liang Mu, Tao Wei, YY Tao, Chang Liang, and XJ Zhang. Design of Medical Image Hardware Acceleration Platform by SDSoC for ZYNQ SoC. *Journal of Image and Graphics*, 8(4):98–106, 2020.
- [NSP⁺15] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015.
- [NSP⁺16] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.
- [NWS⁺20] Tan Nguyen, Samuel Williams, Marco Siracusa, Colin MacLean, Douglas Doerfler, and Nicholas J. Wright. The Performance and Energy Efficiency Potential of FPGAs in Scientific Computing. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 8–19, 2020.
- [Off21] The Met Office. LFRic-Model: A modelling system fit for future computers-Met Office. <https://www.metoffice.gov.uk/research/approach/modelling-systems/lfric>, 2021. [Online; accessed 12-January-2021].
- [OG20] Optimisation-Guide. Xilinx SDAccel optimisations guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1207-sdaccel-optimization-guide.pdf, 2020. [Online; accessed 01-September-2020].

- [OPAM21] Guillermo Oyarzun, Daniel Peyrolon, Carlos Alvarez, and Xavier Martorell. An fpga cached sparse matrix vector product (spmv) for unstructured computational fluid dynamics simulations. *arXiv preprint arXiv:2107.12371*, 2021.
- [Pap12] Michail Emmanouil Pappas. *Parallelisation of Shallow Water Simulation for Heterogeneous Architectures*. PhD thesis, MSc dissertation for University of Manchester, 2012.
- [PFC20] Nuno Paulino, João Canas Ferreira, and João MP Cardoso. Optimizing OpenCL Code for Performance on FPGA: k-Means Case Study With Integer Data Sets. *IEEE Access*, 8:152286–152304, 2020.
- [PKB⁺16] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. *ACM SIGARCH Computer Architecture News*, 44(2):651–665, 2016.
- [Pro20] European Exascale Projects. European Exascale Projects. <http://exascale-projects.eu/>, 2020. [Online; accessed 01-September-2020].
- [PZMM17] Artur Podobas, Hamid Reza Zohouri, Naoya Maruyama, and Satoshi Matsuoka. Evaluating high-level design strategies on FPGAs for high-performance computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.
- [QDL⁺19] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H. Jones. Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8, 2019.
- [Sad75] Robert Sadourny. The dynamics of finite-difference models of the shallow-water equations. *Journal of the Atmospheric Sciences*, 32(4):680–689, 1975.
- [Sca11] Matthew Scarpino. OpenCL in action: how to accelerate graphics and computations. 2011.

- [SEEZ19] Kholoud Shata, Marwa K Elteir, and Adel A El-Zoghabi. Optimized implementation of OpenCL kernels on FPGAs. *Journal of Systems Architecture*, 97:491–505, 2019.
- [SG20] SDSoc-Guide. Xilinx SDSoc development guide. https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/index.html, 2020. [Online; accessed 01-September-2020].
- [SHY14] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):695–705, 2014.
- [SKN⁺16] Kentaro Sano, Fumiya Kono, Naohito Nakasato, Alexander Vazhenin, and Stanislav Sedukhin. Stream computation of shallow water equation solver for FPGA-based 1D tsunami simulation. *ACM SIGARCH Computer Architecture News*, 43(4):82–87, 2016.
- [SMB⁺] Sasa Stojanovic, Veljko Milutinovic, Dragan Bojic, Miroslav Bojovic, Oliver Pell, Michael J Flynn, and Oskar Mencer. Comparing MultiCore, ManyCore, and DataFlow SuperComputers: Acceleration, Power, and Size.
- [SP11] Avinash Sodani and C Processor. Race to exascale: Opportunities and challenges. In *Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture*, 2011.
- [Sve16] Bo Joel Svensson. Exploring OpenCL Memory Throughput on the Zynq. Technical report, Technical Report, 2016.
- [SVK] Leonardo Solis-Vasquez and Andreas Koch. A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software.
- [SVK18] Leonardo Solis-Vasquez and Andreas Koch. A case study in using opencl on fpgas: Creating an open-source accelerator of the autodock molecular docking software. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, pages 1–10. VDE, 2018.

- [SW19] Benjamin Carrion Schafer and Zi Wang. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2019.
- [TM08] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.
- [VH20] Vivado-HLS. Xilinx Vivado SDK. <https://www.xilinx.com/products/design-tools.html>, 2020. [Online; accessed 01-September-2020].
- [VHKF16] Anshuman Verma, Ahmed E Helal, Konstantinos Krommydas, and Wu-Chun Feng. Accelerating workloads on FPGAs via OpenCL: A case study with opendwarfs. Technical report, Department of Computer Science, Virginia Polytechnic Institute & State . . . , 2016.
- [VN14] Mario Vestias and Horácio Neto. Trends of CPU, GPU and FPGA for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, 2014.
- [Vre94] Cornelis Boudewijn Vreugdenhil. *Numerical methods for shallow-water flow*, volume 13. Springer Science & Business Media, 1994.
- [WHU18] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. *Design of FPGA-based computing systems with OpenCL*. Springer, 2018.
- [Wina] Intel-products. <https://www.intel.com/content/www/us/en/products/programmable.html>. Accessed: 2018-08-26.
- [Winb] OpenCL section in Xilinx document. https://www.xilinx.com/support/documentation/sw_manuals/ug1207-sdaccel-performance-optimization.pdf. Accessed: 2019-11-12.
- [Winc] Programming Hardware Functions . https://china.xilinx.com/html_docs/xilinx2019_1/sdsoc_doc/hui1519743141622.html. Accessed: 2022-02-11.

- [Wind] Xilinx-products-devices. <https://www.xilinx.com/products/silicon-devices.html>. Accessed: 2018-08-26.
- [Wine] Xilinx SOC FPGAs board. <https://www.xilinx.com/products/boards-and-kits/ek-ul-zcu102-g.html>. Accessed: 2019-11-12.
- [Xil] Xilinx PCI Express FPGAs board. https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf. Accessed: 2019-11-12.
- [Xil19] Xilinx. Xilinx SDSoC Environment User Guide UG1027 (v2019.1). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1027-sdsoc-user-guide.pdf, 2019. [Online; accessed 01-September-2020].
- [Xil20] Xilinx. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. <https://www.xilinx.com/products/boards-and-kits/ek-ul-zcu102-g.html>, 2020. [Online; accessed 01-September-2020].
- [Xil21a] Xilinx. AXI Reference Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf, 2021. [Online; accessed 12-January-2021].
- [Xil21b] Xilinx. Bare-Metal System Running on Both Cortex-A9 Processors. https://www.xilinx.com/support/documentation/application_notes/xapp1079-amp-bare-metal-cortex-a9.pdf, 2021. [Online; accessed 12-January-2021].
- [Xil21c] Xilinx. SDSoC Development Environment Help. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1207-sdaccel-optimization-guide.pdf, 2021. [Online; accessed 12-January-2021].
- [Xil21d] Xilinx. Vivado Design Suite User Guide. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2020_2/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives, 2021. [Online; accessed 12-January-2021].

- [Xil21e] Xilinx. Vivado Design Suite User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug910-vivado-getting-started.pdf, 2021. [Online; accessed 12-January-2021].
- [Xil21f] Xilinx. Vivado Design Suite User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug903-vivado-using-constraints.pdf, 2021. [Online; accessed 12-January-2021].
- [Xil21g] Xilinx. Vivado Design Suite User Guide Designing with IP. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug896-vivado-ip.pdf, 2021. [Online; accessed 12-January-2021].
- [Xil21h] Xilinx. Xilinx SDSoc programmers Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1278-sdsoc-programmers-guide.pdf, 2021. [Online; accessed 12-January-2021].
- [XO20] Xilinx-OpenCL. Xilinx OpenCL documentation. https://www.xilinx.com/support/documentation/sw_manuals/ug1207-sdaccel-performance-optimization.pdf, 2020. [Online; accessed 01-September-2020].
- [XX20] Chong Xiong and Ning Xu. Performance Comparison of BLAS on CPU, GPU and FPGA. In *2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, volume 9, pages 193–197, 2020.
- [Zoh18] Hamid Reza Zohouri. High performance computing with FPGAs and OpenCL. *arXiv preprint arXiv:1810.09773*, 2018.
- [ZP20] Hanqing Zeng and Viktor Prasanna. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 255–265, 2020.

- [ZPM18] 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*, pages 153–162, 2018.

Appendix A

The Shallow Water Model Source Code

A.1 SWM Nine Kernels Host Source Code

```
//SWM OpenCL Host Code

#include "xcl2.hpp"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <vector>
#include <algorithm>
#include <cstdio>
#include <string>
#include <ctime>
#include <time.h>

using namespace std;

int main(int argc, char **argv) {

    // Grid size
    int M = 64, N = 64;
```

```

int M_LEN = M + 1, N_LEN = N + 1;
int ELEMENTS = M_LEN * N_LEN;
std::cout << "number of ELEMENTS:" << ELEMENTS << endl;

// simulation run iterations
int ITMAX = 4000;

int ncycle;
int i, j;
int err;
float dt = 90;
float tdt = dt;
float dx = 100000.0;
float dy = 100000.0;
float fsdx = (4. / dx);
float fsdy = (4. / dy);
float a = 1000000.0;
float alpha = 0.001;
float el = (N * dx);
float pi = (4.0 * atanf(1.0));
float tpi = pi + pi;
float di = (tpi / M);
float dj = (tpi / N);
float pcf = ((pi * pi * a * a) / (el * el));

std::vector<float> source_p(ELEMENTS);
std::vector<float> source_u(ELEMENTS);
std::vector<float> source_v(ELEMENTS);

/*****
/**OPENCL HOST CODE AREA START****
/*****/

```

```
unsigned int vector_size_bytes = sizeof(float) * ELEMENTS;

//Getting Xilinx Platform and its device
std::vector < cl::Device > devices = xcl::get_xil_devices();
cl::Device device = devices[0];
std::string device_name = device.getInfo<CL_DEVICE_NAME>();

//Creating Context and Command Queue for selected Device
cl::Context context(device);
cl::CommandQueue q(context, device);

//Loading XCL Bin into char buffer
std::string binaryFile = xcl::find_binary_file(device_name, "shallow");
cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
devices.resize(1);
cl::Program program(context, devices, bins);

std::cout << "create kernel init1" << endl;
cl::Kernel kernel1(program, "init1", &err);
auto kernel_init1 = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, float,
float, float, float>(kernel1);

std::cout << "create kernel init2" << endl;
cl::Kernel kernel2(program, "init2", &err);
auto kernel_init2 = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&,
float, float>(kernel2);

std::cout << "create kernel initpc" << endl;
cl::Kernel kernel3(program, "initpc", &err);
auto kernel_initpc = cl::KernelFunctor<cl::Buffer&, cl::Buffer&,
cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&>(kernel3);

std::cout << "create kernel l100" << endl;
cl::Kernel kernel4(program, "l100", &err);
auto kernel_l100 = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&,
```

```

cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&, float, float>(
kernel4);

std::cout << "create kernel l100_pc" << endl;
cl::Kernel kernel5(program, "l100_pc", &err);
auto kernel_l100_pc = cl::KernelFunctor<cl::Buffer&, cl::Buffer&,
cl::Buffer&, cl::Buffer&>(kernel5);

std::cout << "create kernel l200" << endl;
cl::Kernel kernel6(program, "l200", &err);
auto kernel_l200 = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&,
cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&,
cl::Buffer&, cl::Buffer&, float, float, float>(kernel6);

std::cout << "create kernel l200_pc" << endl;
cl::Kernel kernel7(program, "l200_pc", &err);
auto kernel_l200_pc = cl::KernelFunctor<cl::Buffer&, cl::Buffer&,
cl::Buffer&>(kernel7);

std::cout << "create kernel l300" << endl;
cl::Kernel kernel8(program, "l300", &err);
auto kernel_l300 = cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&,
cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&,
cl::Buffer&, float>(kernel8);

std::cout << "create kernel l300_pc" << endl;
cl::Kernel kernel9(program, "l300_pc", &err);
auto kernel_l300_pc = cl::KernelFunctor<cl::Buffer&, cl::Buffer&,
cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&, cl::Buffer&,
cl::Buffer&, cl::Buffer&>(kernel9);

//Creating Buffers inside Device
cl::Buffer buffer_psi(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_p(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_u(context, CL_MEM_READ_WRITE, vector_size_bytes);

```

```

cl::Buffer buffer_v(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_uold(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_vold(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_pold(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_cu(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_cv(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_z(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_h(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_unew(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_vnew(context, CL_MEM_READ_WRITE, vector_size_bytes);
cl::Buffer buffer_pnew(context, CL_MEM_READ_WRITE, vector_size_bytes);

//init1 psi and p
kernel_init1(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),
buffer_p, buffer_psi, a, di, dj, pcf);
q.finish();

//init2 u,v
kernel_init2(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),
buffer_u, buffer_v, buffer_psi, dx, dy);
q.finish();

//initpc uold, vold, pold
kernel_initpc(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),
buffer_u, buffer_v, buffer_p, buffer_uold, buffer_vold,
buffer_pold);
q.finish();

//the main loop

```

```

for (ncycle = 1; ncycle <= ITMAX; ncycle++) {

//l100
kernel_l100(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),
buffer_u, buffer_v, buffer_p, buffer_cu, buffer_cv, buffer_z,
buffer_h, fsdx, fsdy);
q.finish();

//l100_pc update bounds of cu,cv,z,h
kernel_l100_pc(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),
buffer_cu, buffer_cv, buffer_z, buffer_h);
q.finish();

float tdts8 = tdt / 8.0;
float tdtsdx = tdt / dx;
float tdtsdy = tdt / dy;

//l200
kernel_l200(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),
buffer_uold, buffer_vold, buffer_pold, buffer_unew, buffer_vnew,
buffer_pnew, buffer_cu, buffer_cv, buffer_z, buffer_h, tdts8,
tdtsdx, tdtsdy);
q.finish();

//l200_pc update bounds of unew,vnew,pnew
kernel_l200_pc(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),

```

```
buffer_unew, buffer_vnew, buffer_pnew);
q.finish();

//l300
if (ncycle > 1) {
kernel_l300(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),
buffer_u, buffer_v, buffer_p, buffer_uold, buffer_vold,
buffer_pold, buffer_unew, buffer_vnew, buffer_pnew, alpha);
q.finish();

//l300_pc

} else {

tdt = tdt + tdt;
kernel_l300_pc(cl::EnqueueArgs(q, cl::NDRange(1, 1, 1), cl::NDRange(1, 1, 1)),
buffer_u, buffer_v, buffer_p, buffer_uold, buffer_vold,
buffer_pold, buffer_unew, buffer_vnew, buffer_pnew);
q.finish();

}

} //main loop end

q.finish();

} //main function end
```

A.2 SWM Nine Kernels Source Code

```
//SWM Kernel Code
```

```
#define M 64
#define N 64
#define M_LEN 65 //M+1

/*****/
// init1 kernel : Initialize potential pressure and psi
__kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
init1(
__global float *p,
__global float *psi,
const float a,
const float di,
const float dj,
const float pcf
) {
int i,j;
for (i=0;i<M_LEN;i++) {
for (j=0;j<M_LEN;j++) {
psi[i*M_LEN+j] = a * sin((float)(i + 0.5) * di) *
sin((float)(j + 0.5) * dj);
p[i*M_LEN+j] = pcf * (cos((float)(2.0 * i * di)) +
cos((float)(2.0 * j * dj))) + 50000.0;
}
}
return;
}
/*****/
```



```

/*****/
//init 2 kernel :Calculate initial values of wind velocities
__kernel void __attribute__ ((reqd_work_group_size(1, 1, 1)))
init2(
__global float *u,
__global float *v,
__global float *psi,
const float dx,
const float dy
) {
int i,j;
//M_LEN
for (i=0;i<M;i++) {
for (j=0;j<N;j++) {
u[(i + 1)*M_LEN + j] = -(psi[(i+1)*M_LEN+(j+1)]-
psi[(i+1)*M_LEN+j])/dy;
v[i*M_LEN + (j + 1)] = (psi[(i+1)*M_LEN+(j+1)] -
psi[i*M_LEN+(j+1)])/dx;
}}
return;
}
/*****/

/*****/
// initpc kernel: Ubdate the boundray of the wind velocities
//and keep the old version of them
__kernel void __attribute__ ((reqd_work_group_size(1, 1, 1)))
initpc(
__global float *u,
__global float *v,
__global float *p,
__global float *uold,
__global float *vold,
__global float *pold
) {

```

```

int i, j;
for (i=0; i<N; i++) {
u[(0)*M_LEN + (i)] = u[(M)*M_LEN + (i)];
v[(M)*M_LEN + (i + 1)] = v[(0)*M_LEN + (i + 1)];
}

for (j=0; j<M; j++) {
u[(j + 1)*M_LEN + (N)] = u[(j + 1)*M_LEN + (0)];
v[(j)*M_LEN + (0)] = v[(j)*M_LEN + (N)];
}

u[(0)*M_LEN + (N)] = u[(M)*M_LEN + (0)];
v[(M)*M_LEN + (0)] = u[(0)*M_LEN + (N)];

for (i=0; i<M_LEN; i++) {
for (j=0; j<M_LEN; j++) {
uold[(i)*M_LEN + (j)] = u[(i)*M_LEN + (j)];
vold[(i)*M_LEN + (j)] = v[(i)*M_LEN + (j)];
pold[(i)*M_LEN + (j)] = p[(i)*M_LEN + (j)];
}}
return;
}

/*****/

/*****/
// kernel l100: calculate the mass fluxes (CU,CV), potential vorticity (z)
//and fluid surface height (h)
__kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
l100(
__global float *u,
__global float *v,
__global float *p,
__global float *cu,
__global float *cv,
__global float *z,

```

```

__global float *h,
const float fsdx,
const float fsdy
) {

int i,j;
for (i=0;i<M;i++) {
for (j=0;j<N;j++) {
cu[(i + 1)*M_LEN + (j)] = .5 * (p[(i + 1)*M_LEN + (j)] +
p[(i)*M_LEN + (j)]) * u[(i + 1)*M_LEN + (j)];
cv[(i)*M_LEN + (j + 1)] = .5 * (p[(i)*M_LEN + (j + 1)] +
p[(i)*M_LEN + (j)]) * v[(i)*M_LEN + (j + 1)];
z[((i + 1)*M_LEN) + (j + 1)] = ((fsdx * (v[((i + 1)*M_LEN) + (j + 1)] -
v[(i)*M_LEN + (j + 1)])) - (fsdy * (u[((i + 1)*M_LEN) + (j + 1)] -
u[((i + 1)*M_LEN) + (j)])) ) ) / (p[((i)*M_LEN) + (j)] +
p[((i + 1)*M_LEN) + (j)] + p[((i + 1)*M_LEN) + (j + 1)] +
p[((i)*M_LEN) + (j + 1)]);
h[(i)*M_LEN + (j)] = p[(i)*M_LEN + (j)] + .25 * (u[(i + 1)*M_LEN + (j)] *
u[(i + 1)*M_LEN + (j)] + u[(i)*M_LEN + (j)] * u[(i)*M_LEN + (j)] +
v[(i)*M_LEN + (j + 1)] * v[(i)*M_LEN + (j + 1)] + v[(i)*M_LEN + (j)] *
v[(i)*M_LEN + (j)]);

}}

return;
}

/*****/

/*****/
// kernel L100: update the boundray of the mass fluxes (CU,CV),
//potential vorticity (z) and fluid surface height (h)
__kernel void __attribute__ ((reqd_work_group_size(1, 1, 1)))
l100_pc(
__global float *cu,
__global float *cv,

```

```

__global float *z,
__global float *h
) {
int i,j;
cu[(0)*M_LEN + (N)] = cu[(M)*M_LEN + (0)];
cv[(M)*M_LEN + (0)] = cv[(0)*M_LEN + (N)];
z[(0)*M_LEN + (0)] = z[(M)*M_LEN + (N)];
h[(M)*M_LEN + (N)] = h[(0)*M_LEN + (0)];
for (j=0;j<N;j++) {
cu[(0)*M_LEN + (j)] = cu[(M)*M_LEN + (j)];
cv[(M)*M_LEN + (j + 1)] = cv[(0)*M_LEN + (j + 1)];
z[(0)*M_LEN + (j + 1)] = z[(M)*M_LEN + (j + 1)];
h[(M)*M_LEN + (j)] = h[(0)*M_LEN + (j)];
}
for (i=0;i<M;i++) {
cu[(i + 1)*M_LEN + (N)] = cu[(i + 1)*M_LEN + (0)];
cv[(i)*M_LEN + (0)] = cv[(i)*M_LEN + (N)];
z[(i + 1)*M_LEN + (0)] = z[(i + 1)*M_LEN + (N)];
h[(i)*M_LEN + (N)] = h[(i)*M_LEN + (0)];
}
return;
}

/*****/

/*****/
//kernel 1200: claculate new values for u,v,p
__kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
1200(
__global float *uold,
__global float *vold,
__global float *pold,
__global float *unew,
__global float *vnew,
__global float *pnew,

```

```

__global float *cu,
__global float *cv,
__global float *z,
__global float *h,
const float tdts8,
const float tdtsdx,
const float tdtsdy
) {

int i,j;

for (i=0;i<M;i++) {
for (j=0;j<N;j++) {
unew[(i + 1)*M_LEN + (j)] = uold[(i + 1)*M_LEN + (j)] +
tdts8 * (z[(i + 1)*M_LEN + (j + 1)] + z[(i + 1)*M_LEN + (j)]) *
(cv[(i + 1)*M_LEN + (j + 1)] + cv[(i)*M_LEN + (j + 1)] +
cv[(i)*M_LEN + (j)] + cv[(i + 1)*M_LEN + (j)]) - tdtsdx *
(h[(i + 1)*M_LEN + (j)] - h[(i)*M_LEN + (j)]);

vnew[(i)*M_LEN + (j + 1)] = vold[(i)*M_LEN + (j + 1)] - tdts8 *
(z[(i + 1)*M_LEN + (j + 1)] + z[(i)*M_LEN + (j + 1)]) *
(cu[(i + 1)*M_LEN + (j + 1)] + cu[(i)*M_LEN + (j + 1)] +
cu[(i)*M_LEN + (j)] + cu[(i + 1)*M_LEN + (j)]) - tdtsdy *
(h[(i)*M_LEN + (j + 1)] - h[(i)*M_LEN + (j)]);

pnew[(i)*M_LEN + (j)] = pold[(i)*M_LEN + (j)] - tdtsdx *
(cu[(i + 1)*M_LEN + (j)] - cu[(i)*M_LEN + (j)]) - tdtsdy *
(cv[(i)*M_LEN + (j + 1)] - cv[(i)*M_LEN + (j)]);
}}

return;
}

/*****/

/*****/

```

```

//kernel L200_pc: update the boundray of the new u,v,p
__kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
l200_pc(
__global float *unew,
__global float *vnew,
__global float *pnew
) {
int i,j;
for (j=0;j<N;j++) {
unew[(0)*M_LEN + (j)] = unew[(M)*M_LEN + (j)];
vnew[(M)*M_LEN + (j + 1)] = vnew[(0)*M_LEN + (j + 1)];
pnew[(M)*M_LEN + (j)] = pnew[(0)*M_LEN + (j)];
}
for (i=0;i<M;i++) {
unew[(i + 1)*M_LEN + (N)] = unew[(i + 1)*M_LEN + (0)];
vnew[(i)*M_LEN + (0)] = vnew[(i)*M_LEN + (N)];
pnew[(i)*M_LEN + (N)] = pnew[(i)*M_LEN + (0)];
}
unew[(0)*M_LEN + (N)] = unew[(M)*M_LEN + (0)];
vnew[(M)*M_LEN + (0)] = vnew[(0)*M_LEN + (N)];
pnew[(M)*M_LEN + (N)] = pnew[(0)*M_LEN + (0)];
return;
}
/*****/

/*****/
//kernel L300: create old values of u,v,
//and p and the new values for u,s and p for the next cycle
__kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
l300(
__global float *u,
__global float *v,
__global float *p,
__global float *uold,
__global float *vold,

```

```

__global float *pold,
__global float *unew,
__global float *vnew,
__global float *pnew,
const float alpha
) {

int i,j;

for (i=0;i<M_LEN;i++) {
for (j=0;j<M_LEN;j++) {

uold[(i)*M_LEN + (j)] = u[(i)*M_LEN + (j)] + (alpha *
(unew[(i)*M_LEN + (j)] - (2. * u[(i)*M_LEN + (j)]) + uold[(i)*M_LEN + (j)]));
vold[(i)*M_LEN + (j)] = v[(i)*M_LEN + (j)] + (alpha * (vnew[(i)*M_LEN + (j)]
- (2. * v[(i)*M_LEN + (j)]) + vold[(i)*M_LEN + (j)]));
pold[(i)*M_LEN + (j)] = p[(i)*M_LEN + (j)] + (alpha * (pnew[(i)*M_LEN + (j)]
- (2. * p[(i)*M_LEN + (j)]) + pold[(i)*M_LEN + (j)]));

u[(i)*M_LEN + (j)] = unew[(i)*M_LEN + (j)];
v[(i)*M_LEN + (j)] = vnew[(i)*M_LEN + (j)];
p[(i)*M_LEN + (j)] = pnew[(i)*M_LEN + (j)];
}
}

return;
}

/*****/

/*****/
//kernel L300_pc : it is called ones for the first cycle only
__kernel void __attribute__ ((reqd_work_group_size(1, 1, 1)))
l300_pc(
__global float *u,
__global float *v,

```

```
__global float *p,
__global float *uold,
__global float *vold,
__global float *pold,
__global float *unew,
__global float *vnew,
__global float *pnew
) {
int i,j;

for (i=0;i<M_LEN;i++) {
for (j=0;j<M_LEN;j++) {
uold[(i)*M_LEN + (j)] = u[(i)*M_LEN + (j)];
vold[(i)*M_LEN + (j)] = v[(i)*M_LEN + (j)];
pold[(i)*M_LEN + (j)] = p[(i)*M_LEN + (j)];

u[(i)*M_LEN + (j)] = unew[(i)*M_LEN + (j)];
v[(i)*M_LEN + (j)] = vnew[(i)*M_LEN + (j)];
p[(i)*M_LEN + (j)] = pnew[(i)*M_LEN + (j)];
}}

return;
}
```


Appendix B

Block designs

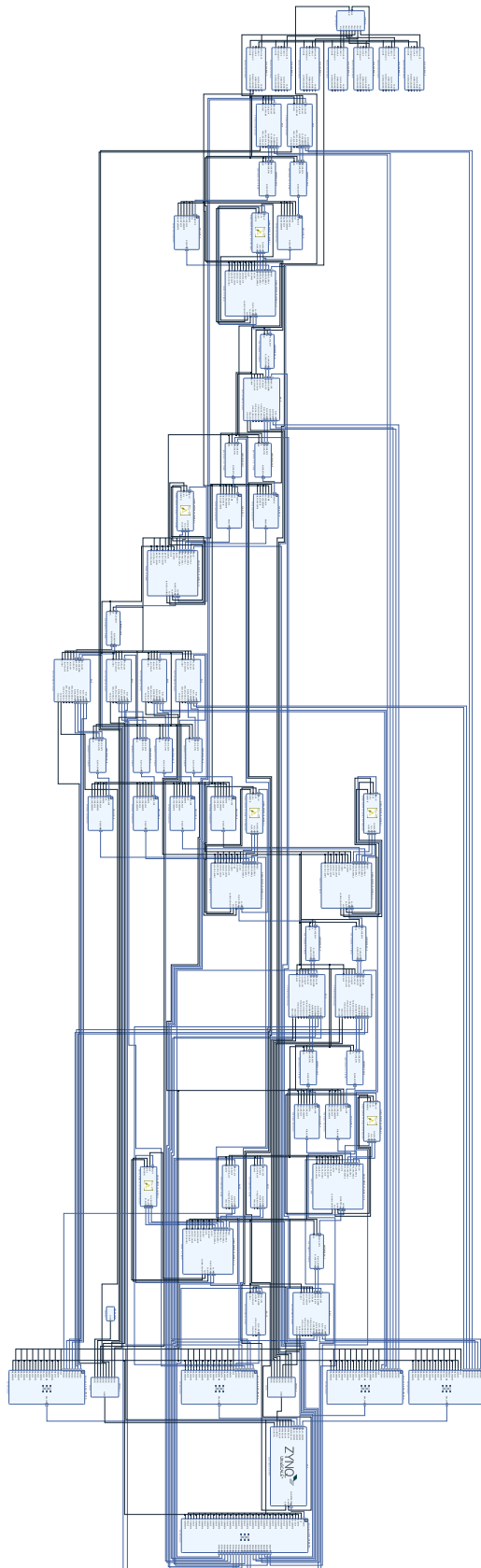


Figure B.1: Overview of the SDSoc C++ six *MatVec* IP blocks system design