Energy-efficient many-core overlay architecture for reconfigurable chips

March 2023

Riadh Ben Abdelhamid

Energy-efficient many-core overlay architecture for reconfigurable chips

Graduate School of Science and Technology

Degree Programs in Systems and Information Engineering

University of Tsukuba

March 2023

Riadh Ben Abdelhamid

Acknowledgements

First and foremost, as a member of the FPGA team laboratory, where I spent the most unforgettable 6 years of my life, first as a research student, then as a master student, and later as a PhD candidate, I am extremely indebted and will be grateful for-life, towards my dear academic advisor, Associate Professor Yoshiki Yamaguchi for his endless support, his infinite kindness, his remarkable advices, and his exceptional leadership skills through which he kept me motivated during the toughest times of this memorable journey.

I am truly grateful to the Ministry of Education, Culture, Sports, Science and Technology in Japan, for granting me the reputable and generous MEXT scholarship, during my 6-year stay in this amazing country.

Many thanks to all of the diplomatic staff of the Embassy of the republic of Tunisia in Japan, for the continuous support of Tunisian students, and for their remarkable efforts to strengthen the academic ties between Japan and Tunisia.

My sincere thanks and appreciations as well, goes to all the hardworking employees of the prestigious University of Tsukuba, who never cease to kindly provide support for the foreign students to fully focus on success and enjoy their study-abroad experience.

I would like to address my special thanks and appreciation to Professor Taisuke Boku, Professor Tsutomu Maruyama, Professor Moritoshi Yasunaga and Professor Osamu Tatebe, for kindly reviewing my PhD application.

Finally, I would love to express my heartfelt gratitude towards my parents for their sincere prayers and heart warming unconditional love, my dear sisters for their kindness and affection, my dear friends and all the special people that contributed to making my life and study abroad experiences more enjoyable during my wonderful journey in Japan.

Abstract

Since the invention of the first programmable CPU (Central Processing Unit), the need for more capable computing machines has been growing by leaps and bounds. This has captured attention of both academics and industries alike, to tackle architecture and physical implementation issues to reach unprecedented levels of computational performance and energy efficiency.

Consequently, new devices emerged to address the shortcomings of conventional processors. Devices such as GPU (Graphics Processing Unit) and FPGA (Field Programmable Gate Array) came to existence and gathered attention as they shifted the previously stagnating computing paradigms from general purpose domains to more application-specific ones, without compromising circuit flexibility. In particular, FPGAs were introduced as chips that are capable to be reconfigured after tape-out and that have a fine-grained nature allowing them to manipulate data at the single-bit level. Nonetheless, the scaling of these VLSI (Very Large Scale Integration) circuits is nearing to hit the power wall that is established by physical laws and as such, the need for energy-efficient devices has been growing larger.

FPGAs are often praised as power-efficient devices, due to the flexible nature of their fabric that allows post-tape-out reconfiguration, and which allows them to address specific computing problems more efficiently. In contrast, these devices still retain a high programmability barrier that even the most seasoned experts may find it difficult to overcome. Worse yet, the development cycle of these devices is lengthy, iterative and cumbersome and complicated designs on top of the largest FPGAs may even take months to see the light.

One interesting way to overcome these shortcomings is the abstraction of the FPGA hardware resources and physical fabric details through what is vastly known as an FPGA overlay. Oftentimes, the chosen level of abstraction and implementation methodology impact the outcomes, in terms of computational efficiency, power dissipation, area utilization and energy efficiency. Among the highest levels of abstraction is a processor-based overlay that exposes a software-like interface, allowing the use of standard or custom HLL (High-Level Language) to program these devices, while providing extremely shorter compilation or configuration times. This abstraction extends to the way in which an FPGA should interact with a host system and access its memory space, in what is called a heterogeneous computing platform.

Consequently, the work in this thesis proposes an **energy-efficient many-core overlay architecture for reconfigurable chips**. The proposed overlay architecture is code-named DRAGON (Dynamically Reprogrammable Architecture for Gather-scatter Overlay Nodes) and aims to address all of the previously stated concerns. In particular, DRAGON is designed through HDL (Hardware Description Language) and its various micro-architecture implementations are specifically tailored to the underlying target FPGA to minimize its resource utilization, maximize its computational performance and boost its energy efficiency.

Besides, DRAGON proposes a custom ISA (Instruction Set Architecture) and is re-programmable through C language, thus yielding a CPU-like compilation time. Furthermore, DRAGON extends the modern RTL (Register Transfer Level) kernel abstraction provided by FPGA tools such as Vitis and proposes a control model from within an OpenCL-based host, to seamlessly interact with its memory space and transcend its boundaries.

Ultimately, the DRAGON architecture condenses multiple parallel processing paradigms in order to minimize the overhead costs intrinsically related to overlays. Consequently, it achieves the highest levels of computational performance and energy-efficiency as proven through an experimental evaluation, using a set of stencil-based benchmarks.

Keywords : FPGA, Overlay, Energy Efficiency, DAE (Decoupled Access Execute), SIMD (Single Instruction Multiple Data), VLIW (Very Large Instruction Word), Many-core, Architecture, 2D,3D,4D Mesh Interconnect, Reconfigurable High Performance Computing.

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence.
ALU	Arithmetic and Logic Unit.
API	Application Programming Interface.
ASIC	Application-Specific Integrated Circuit.
AXI	Advanced eXtensible Interface.
BC	Broadcast Cluster.
BM	Broadcast Memory.
BMC	Broadcast Memory Controller.
BRAM	Block Random Access Memory.
CISC	Complex Instruction Set Computer.
CLB	Configurable Logic Block.
CMOS	Complementary metal–oxide–semiconductor.
CNN	Convolutional Neural Network.
CPI	Cycles Per Instruction.
CPU	Central Processing Unit.
CU	Control Unit.
DAE	Decoupled Access Execute.
DCS	Dual Compute Slot.
DMA	Direct Memory Access.
DNA	Deoxyribonucleic Acid.
DRAGON	Dynamically Reprogrammable Architecture for Gather-scatter Overlay Nodes.
DRAM	Dynamic Random Access Memory.
DSP	Digital Signal Processor.
ECC	Error Correcting Code.
EPR	Effective-to-peak Performance Ratio.
EXACC	EXtreme ACCelerator.
FF	Flip Flop.
FIFO	First In First Out.
FMAC	Floating-Point Multiply-ACcumulate Unit.
FPGA	Field Programmable Gate Array.
FU	Functional Unit.
GM	Global Memory.
GPU	Graphics Processing Unit.

HBM	High-Bandwidth Memory.
HDL	Hardware Description Language.
HLL	High-Level Language.
HLS	High-Level Synthesis.
HPS	Hard Processor System.
IM	Instruction Memory.
ISA	Instruction Set Architecture.
LM	Local Memory.
LUT	LookUp Table.
MAC FPU	Multiply-ACcumulate Floating-Point Unit.
MIMD	Multiple Instruction Multiple Data.
MITRACA	Manycore Interlinked Torus Reconfigurable Accelerator Architecture.
MOSFET	Metal–Oxide–Semiconductor Field-Effect Transistor.
MS	Memory Slot.
NoC	Network on Chip.
OOP	Object-Oriented Programming.
PCIe	Peripheral Component Interconnect Express.
PE	Processing Element.
PE ID	Processing Element IDentifier.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computer.
RTL	Register Transfer Level.
SDP	Simple Dual Port.
SIMD	Single Instruction Multiple Data.
SLL	Super Long Line.
SLR	Super Logic Region.
SoC	System on Chip.
SP	Sustained Performance.
SSI	Stacked Silicon Interconnect.
STA	Static Timing Analysis.
TPP	Theoretical Peak Performance.
URAM	Ultra Random Access Memory.
VHDL	Very High-Speed Integrated Circuit Hardware Description Language.

VLIW	Very Large Instruction Word.
VLSI	Very Large Scale Integration.
XML	Extensible Markup Language.

Part I

Introduction and Background

Chapter 1

Introduction

Since the very first invention of instruction-set micro-processor devices, the quest for increasing their computational performance has never been completed. As such, computer scientists relentlessly investigated novel programming models and paradigms to tackle performance degradation issues encountered by computing devices. Consequently, parallel processing of data has become a hot research topic that triggered a wave of innovative computer architectures and computing approaches, to deal with the ever growing nature of applications.

Since then, parallel processing paradigms such as VLIW (Very Large Instruction Word) [11], SIMD (Single Instruction Multiple Data) [12], and DAE (Decoupled Access Execute) [13] have emerged to target specific bottlenecks preventing mainstream CPUs (Central Processing Units) from reaching their peak computational performance. For example, the SIMD execution model aims to batch-process multiple data with a single control instruction, thus, reducing memory requirements for instructions and removing the redundant hardware control logic, in parallel processing applications such as image processing [14, 15, 16]. In this paradigm, multiple cores or PEs (Processing Elements) apply the same operation (dictated by the instruction) to multiple different data, that are stored into their respective local memories [12]. Besides, the VLIW approach aims to solve control issues in parallel processors by providing a single large instruction, that stacks multiple smaller instruction packets, that are executed in parallel in each core or PE [11]. On the other hand, the DAE approach is quite unique in the sense that it aims at increasing the overall system performance by decoupling the data flow control path from the execution path, usually, through providing a memory interface and low-level programming of two separate instruction domains [13]. The first deals with data movement back and forth between the GM (Global Memory) and the execution cores. The second deals with the execution itself, performed by these cores. Through this scheme, it ensures, to some extent, the removal of data starvation issues, where execution units have to wait for the data to be available, in order to execute operations on them.

These innovative solutions were complemented by the invention of novel ISAs (Instruction Set Architectures) or the addition of modern extensions to the already existing ones. Consequently, new instructions have been constantly added to catch up with the demanding nature of computing applications. In particular, some complete portions of specialized instructions have been introduced to enhance further the performance capabilities of computing devices. For example AVX [17], NEON [18], and VIS [19], among others, were proposed to accelerate multimedia applications.

Nonetheless, architectures based on instruction-sets offer a convenient flexibility that allows to address a wide range of applications. A Turing-complete ISA can solve about any computing problem it may encounter. However, performance and energy-efficiency issues may arise alongside the programming flexibility. Whether ISA-based processing machines are the best to address modern day computing problems is a subject to debate. In fact, these are facing multiple bottlenecks, namely, the memory wall and the power wall and that highlights their limitations. Added to that, the slow down of Moore's Law, that is another issue facing them. In fact, Gordon Moore, who was the Intel company co-founder, has predicted that the amount of transistors that can be condensed in a chip will nearly double after every two years, [20]. The physical proprieties of semiconductor-based chips will soon hit a power wall where the dissipated energy per square millimeter sets a limit on the amount of transistors that may actually fit inside the same area. Certainly, manufacturing technology process known as technology node, is extending the lifespan of Moore's law, through the miniaturisation of transistor size, that is reaching unprecedented levels, that appear to be soon nearing the landmark of a one nanometer.

The quest for better computational performance and improved management of energy requirements, has paved the road to the invention of dedicated computer devices that are more efficient when targeting specific types of computation. Among these, GPUs (Graphics Processing Units) have emerged as the defacto chip for image processing, and are heavily used in gaming arena.

Nowadays, their use case scenarios have been extended to target demanding scientific computing applications, where the more conventional CPU devices, may struggle to perform, as equally well. These devices use smaller and simpler processing cores as compared to CPUs. This allowed GPUs to incorporate a huge amount of computing cores that made them more capable of targeting parallel processing applications in shorter amounts of time and led to the creation of novel programming approaches that challenge even the most seasoned software programming experts.

Nonetheless, GPUs have kept most of the general architectural concepts used to build CPUs, and as such, they are facing most of the issues that have been encountered prior to their existence. In fact, the transistor integration rate is still slowing down while the underlying chips are requiring more and more power. The memory wall is also another heavy-weight issue that must be dealt with. While, modern day GPUs have packed HBM (High-Bandwidth Memory) technology within their chip offerings, still, their architectures intrinsically suffer from inefficient management of the higher memory bandwidth offered by this technology. Certainly, programming tools and compilers evolved to assist extracting the highest possible performance or the lowest energy consumption, however, limitations arise with control-heavy programs, which hinders the capabilities of such devices to underwhelming levels of performance and energy-efficiency.

To optimize their energy-efficiency, computing devices need not only to consume less power, but also to efficiently use that power. To achieve this goal, the sustained computational performance should optimally be equal to the TPP (Theoretical Peak Performance). In modern era of power-hungry applications such as Big Data and AI (Artificial Intelligence), the main limiting factor for such a goal, is unarguably the inefficient flow of data between memories (where they reside) and the execution units (where operations need to be performed on them).

Here, I reached what I believe is the most interesting computing device that has ever been invented to-date : FPGA (Field Programmable Gate Array). These devices have long been similar to white canvas where designers may use HDL (Hardware Description Language) such as SystemVerilog, Verilog or VHDL (Very High-Speed Integrated Circuit Hardware Description Language) to create a custom circuitry that is tailored for a specific kind of computation.

This concept allowed the efficient use of the underlying FPGA structure, to create near-flawless movements of data, use custom-precision data-types, and orchestrate better the overall execution while effectively reducing the energy consumption. Furthermore, the white-canvas-nature of

FPGAs has triggered researchers into proposing efficient reconfigurable computing architectures based on data-oriented paradigms such as data-flow processors [21, 22, 23] and systolic arrays [24, 25, 26, 27].

1.1 Issues and motivation

FPGAs are often praised for their energy-efficiency as compared to the more conventional computing devices such as GPUs and CPUs. Therefore, they offer a valuable asset for researchers and industries alike, either for use as a viable computing device alternative or as a vessel to conduct cutting-edge fundamental theory of computation research.

Where FPGAs really shine, is in the extensive customisation possibilities they offer. For example, the bit width of data can be freely manipulated. As such, it can be extended for a better accuracy [28] or shortened to the bit-level for an improved energy-efficiency and/or performance [29, 30]. Moreover, almost any type of interconnect can be implemented to improve the flow of data between different execution units and/or memories. Examples including tightly-coupled architectures [31, 32], direct and indirect networks [33], different degrees and topologies [34, 35, 36] may be adopted to address about any specific computation need. Furthermore, the execution units can be customised as well. In fact, the implemented circuitry can adopt either a homogeneous model, where all the execution units behave in the same manner and manage the same kind of instructions, or, it can adopt a heterogeneous model where these execution units differ in nature, and may address different data types or different kinds of computation.

Nonetheless, FPGAs have grown in size and complexity and their general architecture is no more the homogeneous matrix of hardware logic blocks it once was. In fact, modern FPGAs have become the sum of small to middle-size FPGAs called dies, that are connected by scarce wires. These dies may be also not completely uniform as they may contain more or less resources of certain kinds compared to others, such as HBM2 memory banks, to say the least, which are mostly condensed in a single die, while absent in the others [37]. Consequently, the low-level programming and the physical mapping of the output circuitry have become an extremely complicated process for hardware designers. These, alongside the lengthy iterations of place-and-route and the time-consuming process of functional verification, are among the key factors that are limiting the practical use of FPGAs. In fact, while there is a soaring demand on FPGAs from industrial and academic organizations alike, these highly-versatile devices remain still far from mainstream adoption.

To simplify the FPGA programming task, HLS (High-Level Synthesis) came to the rescue, as an alternative to the ever complicated and burdensome design process based on HDL (Hardware Description Language). This approach allows programming FPGAs using HLLs (High-Level Languages), such as C or C++. Consequently, the HLS-based methodology has managed to attract designers with a software background, in particular, those with a minimal knowledge of hardware aspects and physical fabric details. Effectively, this approach has dramatically reduced the design and verification times, even when used with the largest FPGAs. However, it still requires all of the lengthy mapping steps, including synthesis, physical implementation and bitstream generation, every time a new design has to be generated.

Another bright solution consists of hiding FPGA hardware resource details and creating a higher abstraction model, using a stacked layered structure, with a configuration interface that allows easier management of FPGA resources and shorter configuration time. This approach is vastly known as an FPGA overlay.

Several overlay architectures have been proposed and implemented on top of FPGAs. A noticeable variety appears in their nature and use cases. For example, some of these are of a fine-grained nature while the others are of a coarse-grained one, even more, some are software-programmable, while others are hardware-configured.

Nevertheless, while most of the previously proposed overlays investigate interesting architectural features and performance benefits, some more complicated topics were rarely covered or remain untapped, such as the integration within a heterogeneous computing platform, the use of more accurate floating-point precisions or even the introduction of custom ISA for reconfigurable devices.

The work in this thesis, aims to bring FPGA closer to the masses while addressing the previously stated concerns and leveraging the general benefits of such a device, through the adoption of the promising overlay approach. In particular, the kind that allows software programming using own custom-designed ISA.

As a result, this thesis entitled "Energy-efficient many-core overlay architecture for reconfigurable chips" presents DRAGON (Dynamically Reprogrammable Architecture for Gather-scatter Overlay Nodes), that is in fact a many-core processor-based FPGA overlay architecture. DRAGON is based on a custom-design ISA (Instruction Set Architecture) and is primarily targeting 64-bit long integer as well as double-precision floating-point computations. In particular, DRAGON aims to provide both an energy-efficient architecture with a relatively high computational performance as well as an easy-to-use methodology to integrate such kind of FPGA overlay into a heterogeneous platform, where a host system, may control the execution of the FPGA overlay using the well established OpenCL framework. Fig. 1.1 shows some of the different research goals encompassed into the thesis title and gives a glimpse on what will be presented and discussed in the subsequent chapters.

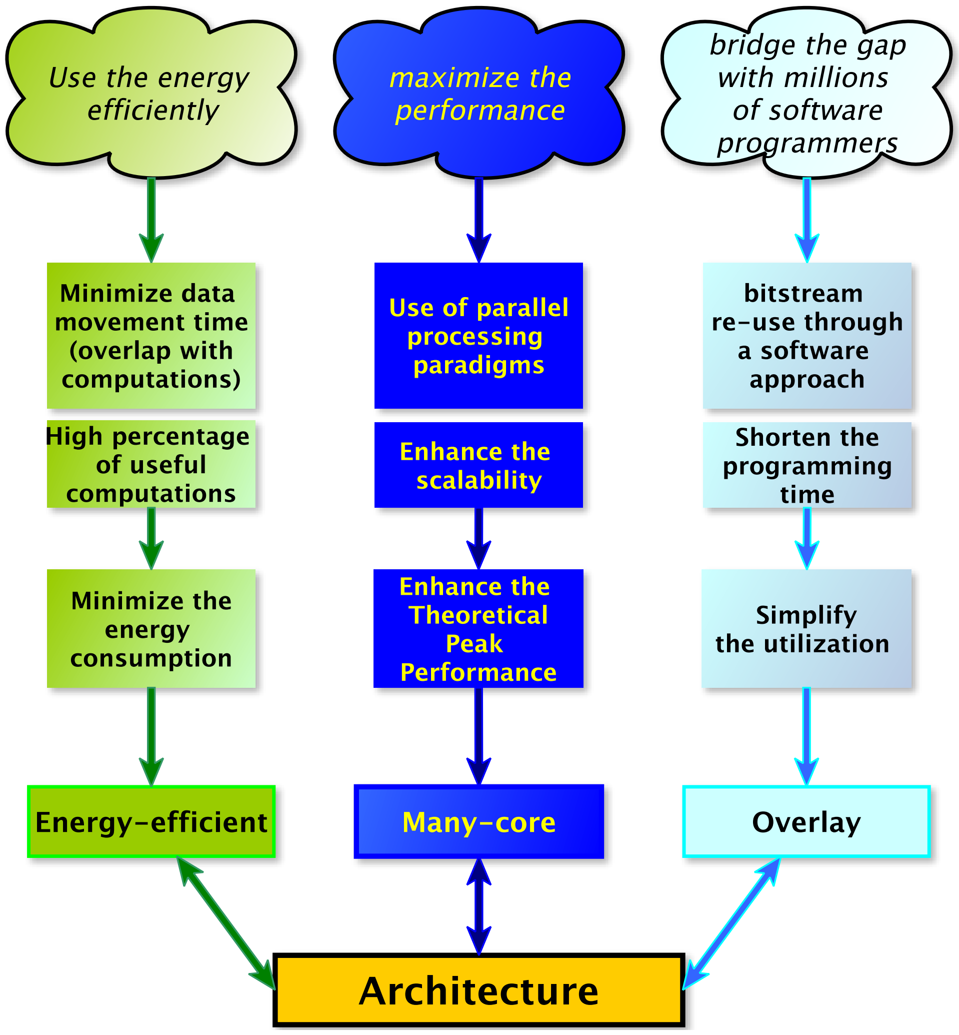

Use the energy efficiently
maximize the performance
bridge the gap with millions of software programmers
Minimize data movement time (overlap with computations)
Use of parallel processing paradigms
bitstream re-use through a software approach
High percentage of useful computations
Enhance the scalability
Shorten the programming time
Minimize the energy consumption
Enhance the Theoretical Peak Performance
Simplify the utilization
Energy-efficient
Many-core
Overlay
Architecture

Figure 1.1: General goals of the work in this thesis as extracted from its title.

1.2 Architecture aspects of FPGAs as compared to other accelerators

FPGAs have different characteristics from other architectures like CPUs and GPUs. This section shows the differences between FPGAs and GPUs, which can be listed as follows.

- **Circuit flexibility and custom granularity** FPGA devices consist of multiple hardware resources and on-chip memories that can be interconnected and configured to realize custom spatial and/or temporal compute architectures. For example, FPGAs can be designed to specifically address computations that require a higher precision (128-bit and higher) or lower precisions (down to the bit level). GPUs focus on thread-level computation with multiple compute units. The optimized design lets a single GPU have a larger number of computing units but rigid on-chip memory interconnects and an architectural hierarchy with fixed data paths and granularity (64-bit, 32-bit, 16-bit, etc.). That's FPGAs are intended to be highly flexible in direct hardware computation compared to GPUs.

- **Reconfiguration** The reconfigurability of FPGAs proposes hardware-level optimization for target applications. It can be realized by maintaining, partly modifying, or entirely re-designing the FPGA-based circuit. In contrast, the hardware circuit of a GPU is fixed.

- **Power efficiency** A large silicon device with high working frequency dissipates considerable power, such as GPUs and FPGAs with HBM. However, FPGAs have proposed hardware-level direct computation, which optimizes data flow and reduces data transfer among processing elements. For example, bit shift operation can be entirely replaced by a wiring connection, which reduces power consumption dramatically.

- **Connection with an external device** FPGAs can directly connect external peripherals, such as memories, buses, and network devices, via high-speed serial interfaces. It enables low-latency data transfer among multiple devices, minimizing data communication administration. Thus, one application of FPGAs will be an acceleration chip that includes a network bridge function toward low latency communication. GPUs also have a special bus like NVLink, but it focuses on only GPU communication, not other devices.

- **Computational performance**

  GPUs were primarily designed to process graphics including image and video data. These devices are extremely efficient by their nature to perform this kind of computation and gradually evolved to further target complex numerical simulations and scientific computing applications. This is mainly backed by their architecture which consists of thousands of concurrently-operating ALUs (Arithmetic and Logic Units), that are coupled with extremely fast memory accesses and large register files. In contrast to FPGAs, these devices can operate above the 1 GHz range of clock frequency. As a result, the raw computational performance of GPUs is significantly higher than that of FPGAs.

- **Programming**

  While complicated when compared to CPUs, programming GPUs is still extremely simpler than programming FPGAs. Compilers can translate HLL (High-Level Language)-based user code into machine executables in an extremely short time. Whereas, FPGA circuits are designed using HDL (Hardware Description Language) or HLS (High Level Synthesis) which require lengthy steps of synthesis, placement and routing until the very last step of bitstream generation.

Nonetheless, despite the high raw computational performance of GPUs, their sustained performance may remain significantly lower in many scenarios, which harms the resulting power efficiency and encourages the exploration of FPGA devices backed by their appealing features that are described earlier.

1.3 Specific target problems and proposed novel solutions

To implement energy-efficient computing circuits running on an FPGA, designers often tend to use the flexible nature of FPGAs that allows them to create specialized computing datapaths, interconnect, and memory hierarchy, which is dedicated to efficiently solve target problems.

While this approach mostly achieves the optimal outcome in terms of computational performance and/or energy efficiency, it remains limited to the specifically targeted problem, and simply changing a few parameters may require redesigning whole portions of the circuit and falls back

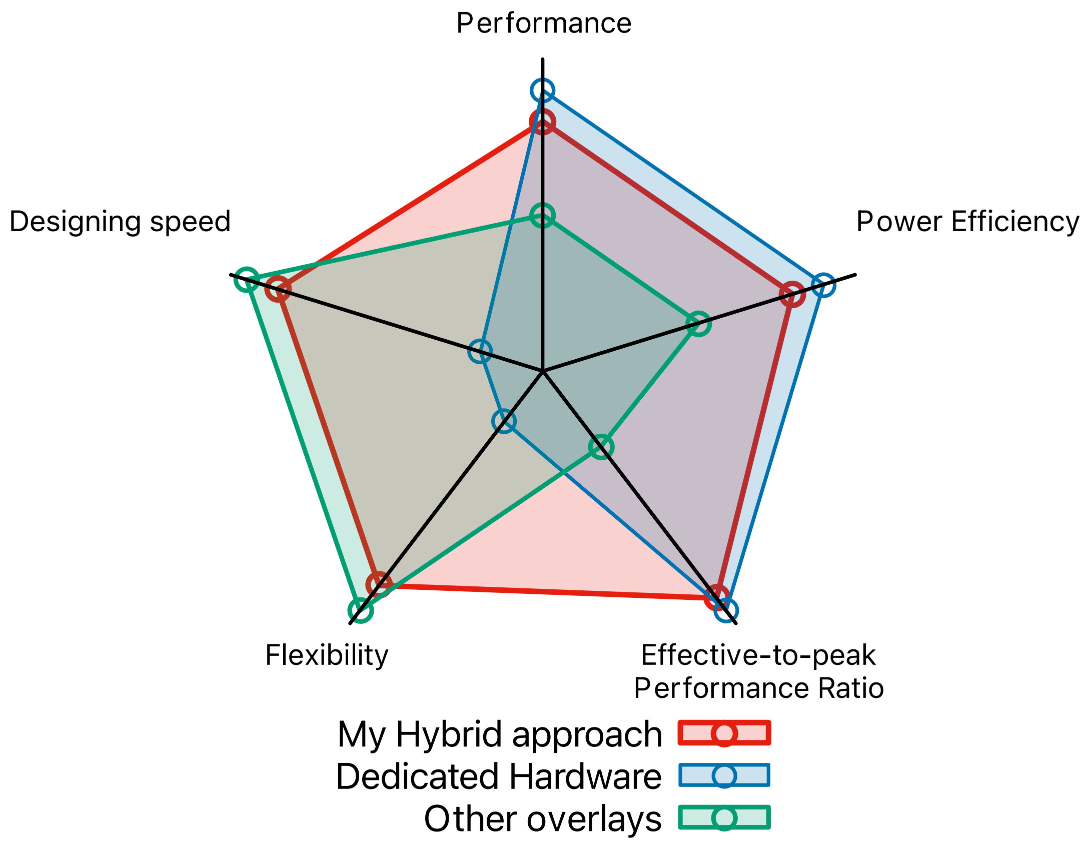


Figure 1.2: Aspects goals of the proposed architecture as compared to problem-specific hardware implementations and other proposed overlay architectures.

to the long and fastidious task of the lengthy hardware implementation. On the other hand, most overlay architectures often sacrifice computational performance and energy efficiency for the sake of the re-usability as well as the convenience that comes with a shorter design process. Here, Fig. 1.2 depicts the goal of this work, which is to try to maximize the benefits from both approaches by adopting an overlay architecture that eases the programming of FPGAs and shortens the design time while adopting specialized architecture and micro-architecture aspects that allow it to maintain relatively high computational performance and energy efficiency as compared to specialized hardware implementations that are restricted to a given problem with fixed parameters.

Moreover, the integration of FPGAs in a computing platform that is controlled by a host system may be challenging and requires low-level knowledge of PCIe firmware and operating system drivers. Here, this work builds upon an existing RTL kernel flow proposed by Xilinx and proposes an enhanced version that allows seamless integration of an overlay within a heterogeneous computing platform. The enhanced approach consists of leveraging the OpenCL task modeling of an FPGA design, by exploiting the built-in PCIe infrastructure and interconnect to send instructions along with data from a host system to the FPGA-based overlay, which allows reprogramming the overlay using the same bitstream file. This leads to the possibility of re-using the same bitstream to target a different set of problems without any need for reconfiguration.

1.4 The attractive reason to use an FPGA overlay

The programming of FPGAs requires hardware detail knowledge and the mastery of circuit designs often at the gate level. As a result, it is an unwanted task by non-hardware experts who prefer the use of more convenient devices such as GPUs and CPUs. An overlay offer an abstract view of the physical fabric of an FPGA which simplifies the programming task at the extent of the chosen abstraction level. To render FPGAs more friendly towards software programmers, a software-programmable overlay adopts the right level of abstraction. To realize this goal, this overlay may adopt a many-core processor-based architecture that enables the efficient use of compute resources while providing a software interface to eventual users.

1.5 Research Goals

The work in this thesis aims to achieve several goals, including the design of an energy-efficient overlay architecture with both performance and versatility in mind. The proposed architecture is expected to provide a software programmable interface through a custom-design ISA and to offer the possibility to be seamlessly integrated within a heterogeneous computing platform. The expected design should be efficiently implemented into the target FPGA to maximize its performance and energy-efficiency. Afterwards, the reconfiguration process consists on recompiling software code and deploying it into the overlay while re-using the original bitstream. Consequently, it can dramatically simplify the process of using FPGAs and shorten their configuration time as well as design cycle, while maintaining the flexibility to address a multitude of computing applications. The **general objectives** may be summarized as follows:

- To bridge the gap between software developers and reconfigurable devices such as FPGAs, through an efficient programming model based on a many-core custom instruction-set overlay architecture, as well as through a simple control approach based on an OpenCL kernel abstraction.

- To maintain the attractive aspects of a task-specific hardware circuit, in terms of energy efficiency and computational performance, while offering the flexibility to target different use cases.
- To adopt architectural (Software and hardware aspects) and micro-architectural (hardware implementation) features that allow the proposed overlay to achieve the previously stated goals, in particular, achieving a relatively high energy efficiency.

The **detailed objectives** may be summarized as follows:

- To survey the state-of-the-art overlay architectures and categorize them in terms of abstraction level , configuration method and application goals.
- To design an Instruction Set Architecture for an energy-efficient and high-performance software-programmable FPGA overlay architecture.
- To provide an efficient method to translate high-level programming code and constructs (that may be based on HLLs such as the C language), into machine code that can be executed by the target FPGA overlay.
- To design an efficient memory architecture and leverage key parallel processing capabilities for the proposed overlay architecture.
- To provide a novel methodology to integrate the proposed overlay within a heterogeneous computing platform.
- To provide and study a preliminary evaluation of the proposed architecture through a baseline FPGA prototype.
- To enhance the scalability, boost the energy-efficiency and tackle the issues related to the FPGA implementation, by optimising the design mapping to the underlying target FPGA.

1.6 Main Contributions

The major contributions of this work may be summarized as follows:

- The design of a custom ISA from a clean slate. The proposed ISA is tightly connected to hardware implementation and assumes a memory architecture consisting of three layered levels. Besides, this ISA supports VLIW paradigm by stacking two instructions into a larger VLIW one. Generally, one packet controls the execution of operations using two operands, while the other packet controls the memory operations such as loads and stores and the data movements such as the scatter and gather operations. The goal of such an approach is the efficient overlapping of data movements with the effective computations.

- The design of a highly modular and energy-efficient many-core overlay architecture that targets reconfigurable chips such as FPGAs. The proposed architecture was given the code-name of DRAGON (Dynamically Reprogrammable Architecture for Gather-scatter Overlay Nodes). The proposed DRAGON overlay architecture adopts a custom-design ISA, and leverages a multitude of parallel processing paradigms such as VLIW and SIMD. It also adopts a particular implementation of the DAE (Decoupled Access Execute) approach.

- A seamless and practical approach to integrate the proposed FPGA overlay architecture into a heterogeneous computing platform, by leveraging the Xilinx RTL kernel implementation methodology. The proposed approach allows the control of processor-based overlays through an OpenCL-based host.

- A versatile PE (Processing Element) architecture that is split into two slots to support the adopted VLIW paradigm. The first slot performs the execution of operations, on either 64-bit double-precision floating-point numbers or 64-bit long integers. The second slot efficiently manages memory operations (Load/Stores) and other data movements (data broadcast, scatter/gather) between the different nodes (PEs) of the many-core overlay.

- A preliminary mapping of the DRAGON architecture into the target FPGA, to collect and study multiple evaluation metrics (Peak performance, power-efficiency, EPR (Effective-to-peak Performance Ratio)), as well as the issues arising from the underlying FPGA implementation (scalability, max clock speed, number of nodes, interconnect degree), that may hinder the capabilities of the overlay. The implementation named Baseline DRAGON overlay achieves an EPR of 89.9% in a 5-point Jacobi stencil computation benchmark which is more than 4 times the EPR obtained with the same benchmark on an Intel Core i9 9900K (20.9%).

- A buffered, point-to-point interconnect architecture, with a base 2D-Mesh topology, that is scalable as well as expandable to higher dimensions.
- A C-based API (Application Programming Interface) that translates mixed C and assembly code into executable machine language, to benefit from existing C compilers instead of designing a custom one.
- Several enhancements of the preliminary DRAGON PE micro-architecture, that include an efficient compact buffering model that reduces the BRAM memory resource utilization by nearly 50% in the proposed many-core overlay and consequently improves its scalability.
- Enhanced micro-architecture FPGA implementations of the DRAGON overlay architecture, named DRAGON2 and DRAGON2-CB (DRAGON2-CB includes the proposed Compact Buffering scheme), that takes into consideration the specific traits of the target FPGA (Alveo U280) [37] and exploits them efficiently. The introduced enhancements unlocked the scalability and the true performance and energy-efficiency on the target FPGA. For example, the enhanced DRAGON2-CB achieves 139.72 GFLOPS, 145.62 GFLOPS and 105.77 GFLOPS, in 2D, 3D and 4D Jacobi stencil benchmarks (double-precision), respectively. Compared to the baseline DRAGON overlay, the enhanced DRAGON2-CB manages to deploy double the amount of PEs, and improves both the sustained computational performance and the power efficiency by more than 4 times.
- A study of the general limitations that face the interconnect and overlay scalability on multi-region FPGAs. Subsequently, a mathematical formulation is provided to obtain the optimal data bus width connecting each GM (Global Memory) bank (that was implemented as HBM (High-Bandwidth Memory) bank) to its corresponding BC (Broadcast Cluster), to maximize the use of inter-die wires without compromising the overlay scalability.

- An in-depth comparative study of DRAGON2 and DRAGON2-CB, on multiple overlay configurations using various N-D interconnect degrees (where N=2, 3 or 4) and various sizes (different amounts of PEs), to evaluate the benefits and costs of the introduced compact buffering model.

1.7 Thesis structure

This thesis is structured into five different parts with a total of eleven chapters:

- **Part I. Introduction and Background**

  Chapter 1: Introduction

  This chapter provides a general introduction to the work in this thesis. This chapter states the issues and motivations behind this work, and follows with a summary of research goals and contributions, as well as a detailed outline of the chapters presented in this thesis.

  Chapter 2: Background

  This chapter presents an overview of the research background. It presents the modern FPGA architectures and their programming methodologies, provides an introduction to the concept of an overlay as well as a literature overview of the previously proposed overlay architectures. Besides, this chapter introduces as well the EXACC [8] and the MITRACA (Manycore Interlinked Torus Reconfigurable Accelerator Architecture) [6, 7] architectures that led to the current DRAGON architecture presented in this thesis.

  Chapter 3: Techniques and principles for energy-efficient FPGA-based many-core overlays

  This chapter presents an overview of techniques and principles that are often involved in the design of energy-efficient digital circuits with an emphasis on instruction-set many-core processor-based FPGA overlays. It reports some of the most important state-of-the-art techniques for lowering the power dissipation as well as the energy consumption in general digital circuits and provides an introduction to the metrics used in this thesis to achieve the goal of a good energy-efficiency, in the proposed many-core overlay architecture.

- **Part II. Software Part: Accelerator softwarization**

  Chapter 4: The DRAGON Instruction Set Architecture

  This chapter presents the details of the proposed custom-designed DRAGON ISA. This chapter illustrates the different categories of instructions and then gives a detailed description of each separate opcode. It also serves as an introduction of the general abstract concepts that can guide the various micro-architecture implementations, including an explanation of the memory architecture and its different levels. This chapter also discusses further extensions of the Instruction Set as well as the use of pseudo-instructions.

  Chapter 5: The Programming Model

  This chapter explains the detailed programming model of DRAGON. The explanation extends to different categories such as the host-side and the FPGA-side. It also explains the extended RTL kernel model used for the control of the overlay and its integration into a heterogeneous computing platform. Moreover, this chapter presents the methodology used to generate binary executable code from HLLs such as the C language. Finally, it discusses functional verification of the DRAGON overlay using Verilator.

- **Part III. Hardware Part: FPGA-based accelerator virtualization**

  Chapter 6: The DRAGON Many-Core-Processor Overlay Architecture: A General Overview

  This chapter introduces the general architecture overview of the proposed DRAGON many-Core overlay as well as its different hardware building blocks. This chapter also serves as a basis for the different micro-architectures, which will be discussed in the chapters that follow.

  Chapter 7: Baseline micro-architecture implementation of DRAGON

  This chapter presents the baseline micro-architecture implementation that serves for the preliminary evaluation. Mainly, this chapter summarizes the work presented in [3].

  Chapter 8: Enhancing the energy-efficiency through DRAGON2 and DRAGON2-CB micro-architecture implementations

This chapter presents an enhanced micro-architecture implementation of the DRAGON overlay and its PE, as well as a novel compact buffering scheme that is used to improve the scalability of the overlay and its interconnect degree. Moreover, FPGA-specific implementation guidelines are presented and discussed in this chapter. Mainly, this chapter summarizes the work presented in [4].

- **Part IV. Results and Discussion**

  Chapter 9: Experiments and Results

  This chapter explains the evaluation methodology as well as the experimental setup.

  The evaluation involves the use of multiple micro-architecture implementations (Baseline DRAGON, DRAGON2 and DRAGON2-CB) and provides a scalability study as well as a comparison between these different implementations, mainly, based on resource utilization, achieved clock speed, computational performance, power efficiency and Effective-to-peak Performance Ratio.

  Chapter 10: Summary and discussion

  This chapter gives a summary of the key differences between the three micro-architecture implementations as well as some of the most important related results. This chapter also discusses the impact of micro-architecture enhancements on programmability. Ultimately, this chapter provides a discussion based on the comparison with related state-of-the-art works.

- **Part V. Conclusion**

  Chapter 11: Conclusion

  This chapter concludes the work in this thesis, mainly, by summarizing the major contributions and achievements.

Chapter 2

Background

2.1 Introduction

This chapter introduces the general architecture of FPGAs and describes the hardware resources populating their physical fabric. Then, it explains their programming approaches using low-level RTL design as well as high-level synthesis methodologies. Then, it gives an overview of the different steps required to implement a design on an FPGA ranging from architecture specification to the last step of bitstream generation. Moreover, techniques related to raising the abstraction level of FPGAs are presented in this chapter, including the concept of overlays. Subsequently, a survey of previous efforts related to FPGA overlay design and a classification of these overlays is presented and discussed. Furthermore, this chapter introduces some of the related works, including the EXACC and the MITRACA overlay architectures that led to the current work.

2.2 General FPGA Architecture

FPGA are reconfigurable chips that allow static configuration as well runtime reconfiguration. Recent FPGA architectures such as Xilinx Ultrascale series provide support for partial configuration as well, where a configuration frame targets specific portions of the device. Basically, configuration allows changing the behavior of a design either statically or during run-time, by loading a stream of binary ’0’s and ’1’s through dedicated pins, to change the ’state’ of the FPGA, including the wiring between its building blocks, or the behavior of these blocks.

Traditionally, thanks to their appealing properties, such as flexibility and abundance of hardware resources, FPGA devices have been exploited to accelerate specific types of computations, that would otherwise struggle to perform well in conventional computing devices such as GPU or CPUs. Among these, applications like DNA (Deoxyribonucleic Acid) sequencing [38, 39, 40, 41, 42], CNN (Convolutional Neural Network) inference acceleration [43, 44, 45, 46], image processing [47, 48, 49, 50], high-precision arithmetic [28], and stencil computing [51, 52] have been accelerated using FPGA devices.

Moreover, several researches investigated the deployment of FPGAs on the cloud [53, 54, 55, 56, 57, 58]. FPGAs are often praised for their energy efficiency, when compared to conventional static architectures such as GPUs and CPUs, which boosts their appealing aspect for deployment as an alternative computing device in data centers [59, 60, 61].

While GPUs and CPUs are flexible computing devices that can be relatively easy to program and build applications for, they have static architectures, where the functionality is fixed in post-production and the data can only flow in the pre-determined paths. In contrast, FPGA devices are capable of splitting and re-routing these paths, in post-production, through the reconfiguration of their interconnect switches. Furthermore, they can also be reprogrammed during runtime through partial reconfiguration or dynamic scheduling of in-FPGA configuration frames targeting its time-multiplexed DSP modules. Consequently, new hardware architecture can be generated every time an FPGA is reconfigured.

Modern-day FPGA devices consist of four basic building blocks, namely, CLBs (Configurable Logic Blocks), hardened blocks (such as BRAM (Block Random Access Memory), URAM (Ultra Random Access Memory), DSP (Digital Signal Processor) [62, 63], etc.), input/output blocks and interconnection resources.

The combination of these reconfigurable hardware resources, offer an adaptive hardware platform that can be reshaped and tailored to address specific computational workloads. Consequently, this allows FPGAs devices to substantially reduce power consumption and increase the computational performance.

In the following subsections, a brief description is given for some of the basic, yet mostly used FPGA hardware resources.

2.2.1 Configurable Logic Blocks

A CLB (Configurable Logic Block) is the base building block in FPGAs and is the main hardware resource that supports the logic reconfiguration aspect of these devices.

For example, in Xilinx FPGAs, it may include multiple-input LUTs (LookUp Tables) and a bunch of registers called FFs (Flip Flops).

The LUT modules can be used to implement basic logic by controlling their output as a function of locally stored input combinations. These modules can also be concatenated to form distributed memories that can hold larger data widths. Moreover, they can be configured as shift registers, multiplexers, or any logic functionality with respect to the available number of inputs [64].

2.2.2 Embedded memories

Modern FPGAs offer multiple different resources that can be used as embedded on-chip memories.

Embedded on-chip memories can be formed using various available FPGA hardware resources such as BRAMs, URAMs [65, 66], LUTs and FFs (registers).

For example, A combination of multiple LUTs can generate memories known as distributed RAMs (Random Access Memories).

FPGAs also embed an abundant number of FFs, that may provide storage for single-bit values or be combined to provide larger memory storage capability, at the cost of increased area utilization.

Besides, URAMs provide a high-capacity on-chip memory storage that can hold up to 256 Kb of data or 288 Kb when counting ECC (Error Correcting Code).

Nonetheless, URAMs are generally less flexible then BRAMs, because of their fixed input width that must be a minimum of 64-bit wide (72-bit with ECC). However, multiple URAMs can be combined to accept larger input widths and build larger memory storage capacity.

On the other hand, BRAMs provide more flexibility than URAMs, by allowing byte-enabled writes and providing support for multiple input widths. They can also be combined to accept larger input widths and build larger memory storage capacity. However, a single BRAM offer significantly lower storage capacity than URAMs (Up to 36 Kb per BRAM).

Finally, FPGA also contain on-board large storage based on DRAM (Dynamic Random Access Memory) while some recent FPGAs offer on-chip HBM memories.

2.2.3 Digital Signal Processors

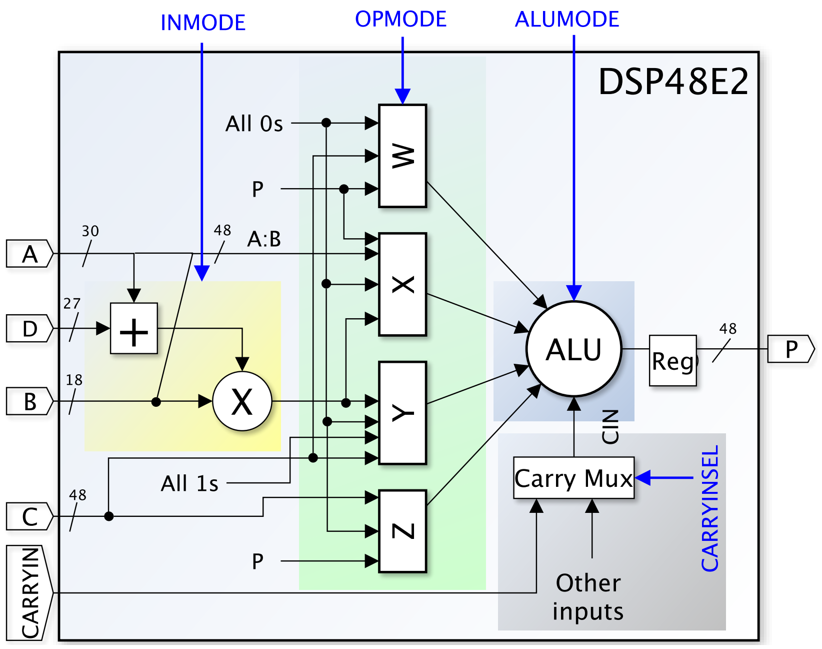


Figure 2.1: General architecture of Xilinx DSP modules.

A DSP (Digital Signal Processor) is one of the most important building blocks of modern day FPGAs. It raises the fine-granularity of FPGAs in specific locations, by combining multiple coarse-grained compute units into a single hardened computing unit.

A typical architecture example of this module is depicted by Fig. 2.1 which depicts the basic blocks of a DSP48E2 that can be found in Ultrascale and Ultrascale+ series of Xilinx FPGAs. The DSP48E2 itself, is a full-fledged computing unit that was investigated by previous works and used as a base processing element to implement complete processing arrays [67, 68]. It has also been used in other innovative ways to implement NoC (Network on Chip) (Network-On-Chip) routers [69].

Besides its capacity to perform additions and multiplications, the DSP48E2 module also embeds a 48-bit inputs ALU (Arithmetic and Logic Unit) that provides support for bitwise operations (such as logical AND, OR, XOR, etc.) and which can be used as well in a packed SIMD manner, effectively performing more operations per clock cycle, using smaller input widths such as a quad 12-bit operands or dual 24-bit operands [63].

Xilinx constantly provides enhancements to its DSP modules. For example, it upgraded the DSP capabilities of the DSP48E1 in the Virtex series [62] to the DSP48E2 in the Ultrascale and

ULtrascale+ series of FPGAs [63]. Among these upgrades, a widened multiplier input width from 25x18 to 27x18.

2.3 Programming Methods

2.3.1 Register Transfer Level

HDL-based design has been the conventional method to program FPGAs. Several HDL programming languages have been widely used by academics and industries alike, namely, VHDL, Verilog and SystemVerilog which provides extensive capabilities compared to its ancestor the Verilog language.

More recently, and with the scaling of VLSI circuits to billions of transistors per chip, the traditional HDL-based programming languages started to slow down design efforts and productivity due to the lack of support for modern programming approaches.

This led to the creation of more capable languages such as Chisel [70] that offers more flexibility and higher abstraction levels, including the use of functional programming and advanced OOP (Object-Oriented Programming) concepts.

The HDL-based approach offers hardware designers means to accurately control their designs behavior at the logic gate level. This level of control yields optimal results and allows relatively fine-tuning the design netlists to optimize the quality of results, with respect to resource utilization, timing and dissipated power.

2.3.2 High-Level Synthesis

HDL-based designs require lengthy, complex and iterative processes of verification, as well as deep knowledge about low-level physical details that are burdensome for non-hardware experts. In a fast-paced technology market, most corporations are looking for means to accelerate their software workloads, without compromising their productivity. In this context, HDL-based design may become outdated as it is reaching its productivity limits. Innovative research efforts such as [71], have introduced an alternative more abstract way to address the FPGA programming methodology, through the HLL friendly tool called HLS. This new approach targets a large community of software designers willing to use FPGAs but not ready to take the leap towards HDL-based tools. As such, it has become possible to use HLLs such as C and C++ and frameworks such as OpenCL to create FPGA-based designs, which dramatically shortens the design cycles and provide substantial productivity boost. Several commercially available tools such as Vivado HLS and Vitis, offered by Xilinx, support HLS-based design and offer programming models that raise further the abstraction level of FPGA hardware.

2.4 FPGA Design Flow

This section presents the traditional design cycles when programming FPGA circuits.

2.4.1 Architecture Specification

At first, an FPGA designer must define the specification of a given problem in a concise manner that would consequently lead to the high-level description of the eventual solution to that problem, or in other words, the circuit architecture. Often time, the original problem is split into smaller ones that may be addressed separately, in a divide-and-conquer manner. Almost always, the problem definition consists on providing a plain description that is understandable in human language and which states clearly and concisely the requirements and constraints of a certain FPGA design. Later on, it is the task of hardware design architects to turn this description into high-level schematics of the target solution. These schematics are a mean to get the original human-language specifications closer to the final hardware implementation of the desired circuit.

2.4.2 Hardware Design and optimization

Defining the problem and its architectural specifications leads to the next step in the design cycle, that is the translation of these specifications into a circuit description, either using an HLS-based approach or a HDL-based one. This step includes sub-steps of optimizations related to the specification constraints such as the desired operating clock frequency, the peak performance to be achieved and the limitations on power dissipation or resources utilization.

During the hardware design phase, it is often common to iterate into this same cycle, while introducing several optimizations that aim to achieve, to a certain extent, the preliminary goals such as reducing the power dissipation of the design, improving its speed or even minimizing its area.

2.4.3 Functional Verification

It is often common that misinterpretations of the original problem happen during the hardware design cycle. Consequently, every designer must ensure that the hardware description he provides reflects correctly the desired behavior stated by the problem specification. In order to achieve this goal, functional verification must be conducted. In this step, test scenarios often called test-benches, provide some constrained random inputs that are fed to the design under test, in a way that mostly reflects the expected design inputs. The goal of these tests is to ensure that the system provides the same expected outputs as those stated by the original problem specification. This design step is among the most important because it may revert the design cycle back to previous steps. Oftentimes, this is also the most effort- and time-consuming section of the entire design flow.

2.4.4 Synthesis

Synthesis is the step in which a given hardware design description is translated into an actual circuit made of basic logic gates, on-chip memories and wires that connect them all. Often, hardware compilers called synthesizers, may introduce further optimizations, in which, they may remove redundant signals and logic, or at the opposite, duplicate them, for example, to solve fan-out issues. The remaining subsequent steps of the FPGA design flow belong to a larger embodying step called the FPGA implementation.

2.4.5 Technology mapping

At this step, the netlist generated after synthesis will be mapped into logic elements that actually exist on the target FPGA, such as LUTs and DSPs. Several transformations are conducted by the implementation tools to allow this mapping.

2.4.6 Placement

The Placement step consists of reserving a physical hardware resource in a certain location on the FPGA to the equivalent technology-mapped netlist resource. Placement can be timing-driven, in which case, the wiring distance between the hardware resources will be accounted for, in particular, for critical paths, at the eventual cost of increasing the wire length of the remaining less critical paths.

2.4.7 Routing

The routing process consists of establishing the connections between the previously allocated hardware resources during the placement step. This step involves the use of interconnection resources available on the target FPGA such as switches and multiplexers, or through hardware resources such as LUTs. Alongside placement, routing is among the longest two parts of the entire FPGA design flow, and despite the dramatic advances on both software algorithms and hardware computing performance, these two steps remain the bottleneck of the whole design process.

2.4.8 Static Timing Analysis

STA (Static Timing Analysis), consists of checking if timing requirements are met in all the paths of a given design netlist. It may be conducted, after synthesis, placement or routing steps. Typically, the STA checks verify that there are neither negative slacks nor violations on setup and hold times. The successful STA verification leads to timing closure, which indicates that a given design is capable of operating at the desired clock speed.

2.4.9 Bitstream Generation

Here comes bitstream generation, which is the final step that concludes the design flow and creates the FPGA design image. A bitstream is the FPGA configuration stream of '1's and '0's, that maps the hardware image on FPGA by configuring its interconnection switches and logic blocks. Basically, FPGAs are reconfigurable devices because they allow the "rewiring" of their hardware resources in post-production, by downloading a new bitstream into their configuration memory. This memory holds the logic state of the desired FPGA circuit when it is powered-on.

2.5 FPGA Overlay Architectures

The general idea of overlay abstraction as well as the particular case of processor-based FPGA-overlays is explained in this section. Furthermore, a survey of the state-of-the-art previously proposed overlays will be subsequently presented.

2.5.1 The motivation behind FPGA Overlays

FPGAs are reconfigurable devices that allow the implementation of different circuits based on different configuration bitstreams. The traditional method of programming such devices is based on hardware description languages such as Verilog which allow the translation of high-level architecture definitions into hardware circuits. While this approach yields the best outcome for a given set of design goals such as performance, energy efficiency or area utilization, it remains extremely difficult for non-hardware experts and requires deep understanding of FPGA physical fabric details. The more leading-edge approach of programming FPGAs consists of using HLS (High-Level Synthesis) that simplifies the design task by providing more powerful software constructs to translate the architecture specification into a circuit netlist. Nonetheless, both approaches still require long iterations of architecture definition as well as netlist generation and implementation on a given target FPGA. Moreover these two programming approaches fall back to the lengthy FPGA compilation times, mainly caused by the placement and the routing steps, during the implementation phase. Worse yet, failure to meet certain constraints such as speed, area or power, requires incremental iterations of the same lengthy steps towards achieving the design goals.

It is no secret that all of the previously stated steps make the FPGA design cycle as a whole, lengthy and cumbersome. In particular, placement and routing require tremendous computational effort and the corresponding processing duration grows considerably with the increase in design complexity or/and FPGA size. This repels the vast majority of software designers and consequently keeps FPGAs far away from mainstream adoption.

Arguably, the most interesting way to abstract FPGAs hardware details and ease their utilization is to use an overlay architecture. The general approach of an overlay is to provide one or more additional layers that hide the low-level details of the FPGA and allow a more simplified programming model. Oftentimes, the underlying abstraction level of such an overlay depends on several trade-offs that may heavily impact the design goals. Generally, the higher this abstraction level, the less optimal the circuit outcome, in terms of speed, area and/or power dissipation. To reduce the impact of such an abstraction, careful design methodologies and guidelines must be strictly followed. The next chapter provides insights into some examples of these guidelines, towards the software/hardware co-design of an energy-efficient many-core overlay architecture.

2.5.2 State-of-the-art FPGA Overlays

Table 2.1: Review of some previous parallel processing overlays [3].

Ref	**Year**	**Name**	**FPU**	**ALU**	**Topology**
General-purpose					
[72]	2012	reMORPH	None	variable	2D Mesh
[73]	2013	TILT	32-bit	None	Crossbar
[74]	2015	SIMD-Octavo	None	36-bit	SIMD Lanes /Mesh
[75]	2016	GRVI	None	32-bit	2D Torus
[76]	2019	2GRVI	None	64-bit	2D Torus
Application-specific					
[52]	2010	SCMA	32-bit	None	2D Mesh
[51]	2014	SSA	32-bit	None	1D Torus x 1D Mesh
Hybrid approach					
Ours	2020	DRAGON	64-bit	64-bit	N-D Mesh/Torus

Table 2.2: State-of-the-art spatially configured vs time-multiplexed overlays [5]

Spatially configured		**time-multiplexed**	
Year	**name**	**Year**	**name**
2006	QUKU[77]	2011	Heracles[78]
2010	IF[79]	2011	CARBON[80]
2011	VDR[81]	2012	reMORPH[72]
2012	ZUMA[82]	2016	GRVI Phalanx[75]
2015	DSP-based[67]	2017	MIPS Overlay[83]

Thanks to their appealing advantages, FPGA-overlays represent a hot research topic where several ideas have been investigated and proposed. A survey about FPGA-based overlays was proposed in [5] and based on their FUs (Functional Units) run-time configurability, it suggested a classification of these overlays as being either time-multiplexed or spatially configured. When the FUs can dynamically adapt their behavior during run-time, the corresponding overlay belongs to the time-multiplexed category, otherwise, it is called a spatially-configured overlay. Some examples of state-of-the-art overlays in both these two categories are presented in Table 2.2 based on the survey proposed in [5]. Another detailed classification based on application flexibility is given by Table 2.1.

ZUMA [82] is an example overlay that is spatially-configured. This is an open-source overlay designed for portability. To achieve this goal, it proposes a virtual FPGA with a fine granularity on top of the commercial FPGA and which is pre-compiled with the implementation tools provided from the FPGA vendor. This allows ZUMA to abstract low-level details of FPGAs by acting as a compatibility layer that provides a standard physical view regardless of what lies beneath it. Nevertheless, the overhead cost of implementation area limits its use for large scale designs.

Besides, overlays that have the ability of dynamically changing their functionality or allow hardware reconfiguration during run-time belong to the category of time-multiplexed overlays as per the classification proposed by [5].

More than a few tens of billions of transistors may be packed into current large-sized FPGAs, allowing them to provide a million or more LUTs and thousands of DSP blocks along with hundreds Mb of on-chip memory storage capacity. Hence, these features combined offer a convenient platform with a sufficient size to implement relatively large arrays of time-multiplexed overlays in the form of 32-bit or 64-bit instruction-set processors. Consequently, this motivated the investigation of such a possibility and has led to interestingly innovative many-core processing systems examples. Among these, an FPGA overlay code-named GRVI Phalanx [75], that is a massively parallel many-core processor based on an extremely small footprint 32-bit RISC-V PEs that operate at 375 MHz and use about 320 LUTs with a reported total of 400 PEs that were implemented on a Xilinx KU040 (Kintex Ultrascale) FPGA device.

A seemingly more recent update of the GRVI Phalanx project, code-named 2GRVI Phalanx [84] uses a larger FPGA device that is equipped with HBM memories and succeeded to deploy 1332 PEs that are based on a 64-bit version of the RISC-V ISA. These PEs maintain a small footprint of just 400 LUTs.

Nonetheless while both the 32-bit [75] and the 64-bit [84] versions of these many-core processing systems provide the possibility of adding custom accelerators, their current implementations lack the support for floating-point operations.

Besides, another interesting overlay that targets area reduction is the reMORPH overlay [72] that has a remarkably efficient resource utilization. In fact, this overlay is able to target relatively small FPGAs thanks to its small FUs (Functional Units) that use, each, a single DSP, three BRAMs, 41 FFs and 196 LUTs. For example, fourty tiles were deployed on a relatively small-sized Xilinx FPGA (Spartan 6). reMORPH uses the internal structure of the DSP module to map its proposed 5-stage ALU which allowed it to reach a clock speed of 400 MHz.

Generally, floating-point capable PEs tend to consume significantly larger amount of hardware resources as compared to those uniquely supporting integer operations. This is particularly true in high precisions where mantissas are larger and their multiplications consume more area which in turn negatively impacts the operating clock speed. To avoid the increased area cost and save sufficient amount of resources to deploy a larger number of PEs, most of the reported works focused on supporting integer-only operations. While this is convenient from the view point of implementation, it narrows the domain of applications that can be addressed by the underlying integer-only overlay. For example, scientific computations require the use of real numbers that are approximated in computers with adequate precision, through the floating-point representation. This kind of representation requires specific hardware to perform basic operations on the underlying data.

To address this requirement, the work in [52] proposed a systolic computational-memory array (SCMA) that ditches integer operation support to provide dedicated support for floating-point operations along with a sufficient flexibility through a minimal programming capability. The proposed architecture targets numerical simulations based on finite difference methods. However, the support for floating-point calculation remains limited to 32-bit single-precision data which seems to be a design compromise between target applications and underlying hardware (to save more resources). The overlay proposed in [52] managed to deploy 192 PEs that can operate at 106 MHz on an Altera Stratix II FPGA.

Other research such as the work in [51] proposed the Scalable Streaming Array (SSA) that is a systolic architecture scalable to a multi-FPGA system. SSA cascades multiple Pipelined-Stage

Modules (PSMs) that can perform stencil calculation for each iteration and are programmed using a domain specific language. Nonetheless, both the works in [52] and [51] lack support for integer-based operations and their support for floating-point data remains limited to 32-bit single-precision.

Compared to the state-of-the-art proposed overlays, the work in this thesis aims to leverage benefits from architectures and hardware implementation techniques that are specific to a given application domain, while maintaining sufficient flexibility through a rich and extendable custom instruction set. The proposed instructions primarily offer support for both 64-bit long integer and double-precision floating-point numbers, through separate execution units from a hardware micro-architecture perspective.

2.5.3 The case of custom processor-based FPGA Overlay

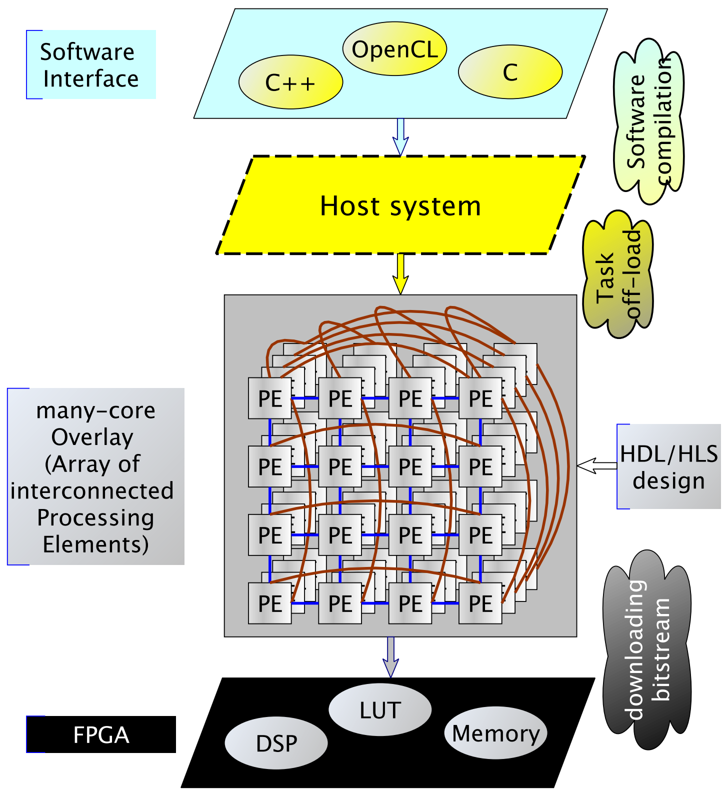


Figure 2.2: FPGA abstraction through the proposed overlay.

A processor-based FPGA overlay follows a special kind of hardware abstraction through the adoption of a software programming layer. This kind of overlay belongs to the time-multiplexed category.

In general an overlay is built using multiple layers that provide gradual abstraction of FPGA hardware resources with the aim to facilitate their programming. An example overview of such a layered abstraction of the physical FPGA is depicted by Fig. 2.2.

At the bottom layer resides all kind of hardware resources that constitute the FPGA fabric.

A processor-based overlay adds a layer of PEs that can perform various operations on input data by executing decoded software instruction. This creates a "softwarized" version of the FPGA where a designer can safely ignore the implementation details below this layer. Here, my proposed overlay architecture implements this layer as a many-core array of PEs that can be interconnected through a given degree and topology to exchange the data between each other in a direct manner. The design of this array of PEs can be performed with the help of HDL (Verilog, SystemVerilog, etc.) or HLS (C, C++) languages. Nonetheless, HDL-based design often provides the best quality of results that ensure optimal resource utilization, fast operation and reduced power dissipation.

On the top of this many-core layer, an interface layer may provide a bridge to communicate with a host system to download non-processed data and software instructions into the FPGA fabric or store back the results of data processing into the host memory. This can be realized through a PCIe (Peripheral Component Interconnect Express) interface with the help of low-level firmware programming, or just by efficiently leveraging the infrastructure provided by the FPGA vendor.

Using such an abstraction, an FPGA overlay may be controlled from a host system by simply offloading tasks using the created host-FPGA interface. These off-loaded tasks are written using instructions that can be interpreted by the many-core layer of the overlay.

A high-level C, C++ or OpenCL-based program can be compiled on the host to provide an executable that allows a host CPU to offload tasks in forms of specifically targeted instructions to the FPGA-overlay. The high-level program also can allow the host system to move data back and forth to/from the FPGA overlay.

Ultimately, this thesis proposes a software-programmable overlay that can be reprogrammed using custom instruction-set opcodes that are abstracted as prototype functions and inlined in a standard C program. The proposed overlay can also be controlled from a host system through an OpenCL program.

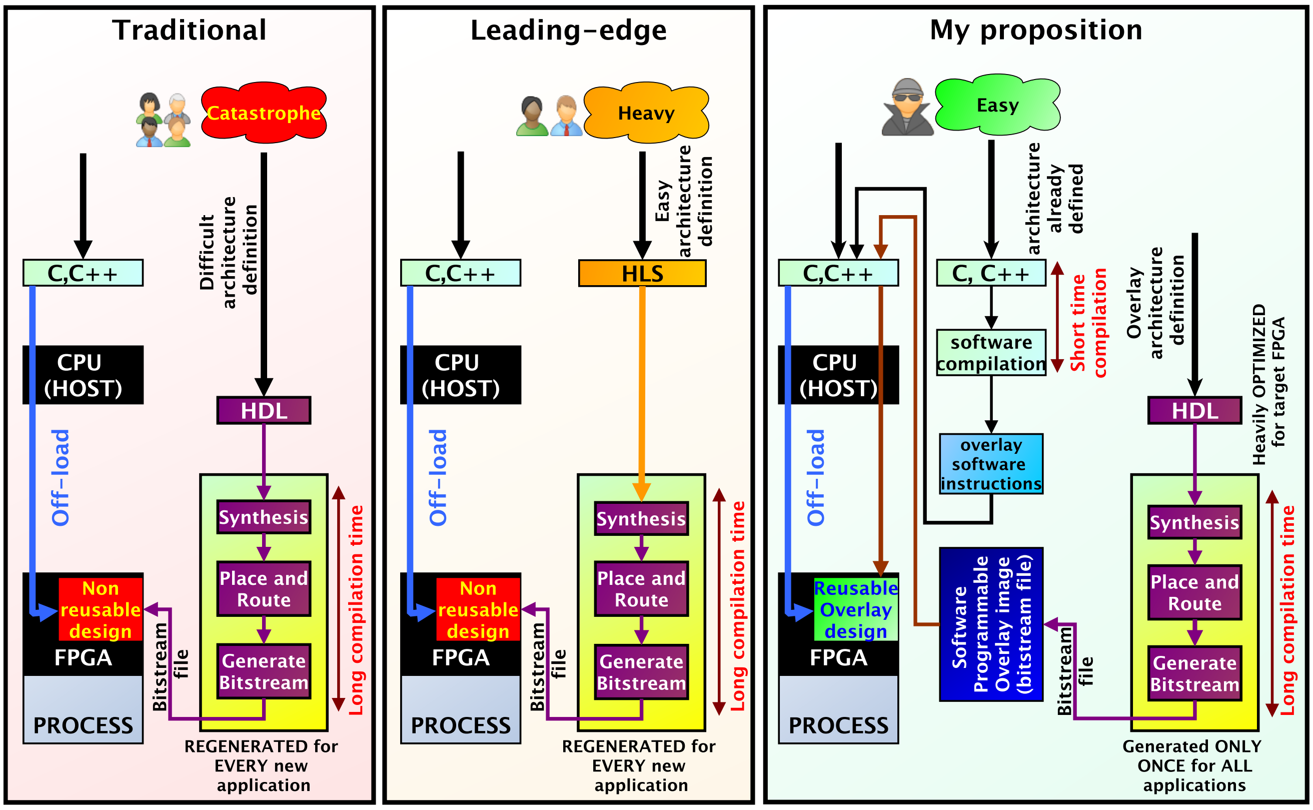


Figure 2.3: Design complexity using different hardware design approaches.

2.6 Enhancing the programming model of FPGAs through software-based re-usability

Fig. 2.3 illustrates the traditional and leading-edge approaches of programming and controlling FPGA devices, as well as my proposed simplified approach.

Traditionally, reconfigurable devices such as FPGAs are programmed using HDL languages (VHDL, Verilog, SystemVerilog, etc.) which offer an abstract description of desired circuit functionality through careful definition of concurrent processes and individual signals. While this methodology often provides the most optimal outcomes in terms of area, power dissipation and computational performance, it is beset with serious design concerns. For example, the complex nature of low-level architecture definition requires deep understanding of hardware resources on the target devices as well as expertise in the back-end implementation including gate-level behavior and physical mapping. Oftentimes, this unnecessarily stretches the design cycle time and slows down the time-to-market for many products. Besides, the implementation itself is lengthy and iterative as it requires multiple synthesis, placement and routing rounds until the design achieves its specified goals.

On the other hand, the leading-edge approach consists of ditching HDL languages for the designer-friendly HLLs such as C and C++. This approach shortens the design cycle duration thanks to a more elevated abstraction of the underlying hardware and the use of more powerful constructs offered by HLLs. As such, this approach allows for easier problem definition through high-level descriptions of circuit state without diving into the details of low-level logic. While this approach significantly shortens the design time, it falls back to the pitfalls of HDL-based designs as it requires to re-apply all the lengthy steps of synthesis, placement, routing and bitstream generation, possibly multiple times until design goals are met.

Vendors such as Xilinx offer a powerful approach of controlling FPGA devices , through RTL kernel (for HDL-based designs) or HLS kernel (for HLS-based designs) abstractions, which provide all the infrastructure of host-based control and PCIe-based communication between a host computer and an FPGA. Despite facilitating the control and communication tasks, these approaches do not directly allow design re-usability and changing even the smallest parameters would require re-implementing a modified design and generating a new bitstream before loading it again into the FPGA device.

Here, my proposition consists of two steps. The first step would be to design a software-programmable overlay architecture using the traditional HDL approach for extracting the best outcomes in terms of area, speed, power efficiency and performance. The second step consists of packaging the overlay using a compatible interface with the RTL kernel model, which allows OpenCL-based data and instructions transfers (from a host), without requiring to reload a new bitstream, resulting in a re-usable and significantly simpler programming model, which will be explained with further details in Chapter 5.

2.7 EXACC Architecture, a base model for MITRACA and DRAGON

The work in this thesis is originally inspired by the EXACC (EXtreme ACCelerator) architecture [8] and the MITRACA (Manycore Interlinked Torus Reconfigurable Accelerator Architecture) [6, 7].

Table 2.3: Comparison between DRAGON (this work), MITRACA [6, 7] and EXACC [8] architectures.

	DRAGON (this work)	**MITRACA[6, 7]**	**EXACC [8]**
VLIW parallelism	Yes	Yes	No
SIMD parallelism	Yes	Yes	Yes
Fused Mul-ACc	Yes	Yes	No
Interconnect degree	Up to 4D	3D	2D
ISA	DRAGON ISA designed from scratch	MITRACA ISA designed from scratch	based on GRAPE-DR [85]
Control Method	custom Control Unit (unified with the DRAGON ISA)	MITRACA ISA separate CPU	based on separate CPU
PE architecture	Register-Memory (allows operations on) broadcasted data)	Register-Memory (allows operations on) broadcasted data)	Register-Register (Load-Store)
Largest PE count (FPGA-based)	288	256	16
Data Broadcast (from BM)	through a dedicated controller	through a dedicated controller	direct connection
Data collection (to BM)	direct connection	through sampler and controller	direct connection
Source of data (that is stored to BM)	RF or LM or Comm. buffers or ALU or FPU	float RF or integer RF	RF only
Source of data (that is scattered to other PEs)	RF or LM or Comm. buffers or ALU or FPU	float RF or integer RF through Comm. register	RF only through C. register
Broadcast dimension (Broadcast Block dim)	2D	1D	1D

The EXACC architecture had been discussed in the feasibility study of the Japanese flagship computer system [86]. The proposed architecture was evaluated on a software simulator [87] and a hardware emulation by FPGA [8]. In [8], an FPGA emulated a small-size EXACC that includes a 4x4 array of PEs connected by a 2D-Mesh network topology. The underlying ISA (Instruction Set Architecture) was based on the GRAPE-DR architecture [85].

In 2018, the MITRACA architecture was recreated based on the EXACC architecture and inherited the two-part aspect of the overall design consisting of an Accelerator and Controller parts. The skeleton framework of MITRACA adopted the same concept as the EXACC while enabling the computation of multiple operands, including communication registers that stored inputs directly from each neighboring PE. Besides, a custom-design ISA was developed from scratch, which proposed a completely redesigned programming model with an aggressive VLIW approach explicitly exposing the parallelism to the programmer. Furthermore, the micro-architecture implementation of MITRACA deployed a considerably larger amount of PEs compared to the work in [8] (256 PEs against just 16). While the larger FPGA target allowed this level of scaling, the architecture's scalability was the main reason for achieving that milestone.

On the other hand, an early version of the DRAGON architecture was proposed in [88]. However, the underlying controller architecture had not been determined yet. Besides, the overlay programming was much more complicated as its PE micro-architecture contained two ALUs and two FPUs, which was reduced to one for each in subsequent iterations in [3, 4].

Ultimately, the DRAGON overlay architecture proposed in this thesis, was inspired by the work in [8], and is a continuation of the previous research efforts of [6, 7, 88]. The DRAGON ISA was completely redesigned and the Controller part ditched the master processor previously adopted in EXACC and later in MITRACA, to implement a custom CU (Control Unit) that communicates with the sequencer to issue the SIMD instruction streams. Moreover, it maintained the possibility of performing operations directly on operands coming from the BM (Broadcast Memory) while improving the original broadcast block architecture by proposing a more modular and scalable broadcast cluster architecture, that consists of 4x4 arrays of PEs and broadcast memories consisting of 16 banks, each. DRAGON creates pairs of PE, BM bank and implements the broadcasting feature on a 2D block instead of 1D block in the original EXACC and MITRACA architectures. Better yet, the broadcasting feature is implemented in a two-level approach that allows to implement two types of broadcast using the base ISA and may be extended to implement custom broadcasting models using further ISA extensions.

Table 2.3 summarizes the notable differences between EXACC [8], MITRACA [6, 7] and DRAGON [88, 3, 4] architectures.

2.8 Summary

This chapter explains mainstream FPGA architecture and programming methodologies, including HDL-based and HLS-based design approaches. It also introduces the concept of FPGA overlay and the necessity for such an approach to bridge the gap between non-hardware experts and the adoption of FPGA as a mainstream computing device. This chapter also summarizes the state-of-the-art FPGA overlays and categorizes them based on multiple architecture and micro-architecture properties. Furthermore, this chapter discusses the case of a custom processor-based overlay and the way to enhance the programming model of FPGAs as an eventual alternative to the shortcomings of previously proposed works. Finally, this chapter introduces the DRAGON overlay and provides a brief summary of the origins of DRAGON, including the MITRACA and EXACC architectures.

Chapter 3

Techniques and principles for energy-efficient FPGA-based many-core overlays

3.1 Introduction

When the first computer chips came to existence, the main focus was to increase their computational performance through incremental innovative approaches. Nonetheless, the increase in circuit sizes and the miniaturization of manufacturing technology nodes has driven computer chips to hit a power wall where the heat dissipation from electrical power consumption has become an inhibiting factor towards further scaling of the number of transistors that may fit into a single die. As a result, energy efficiency has become a serious concern for chip manufacturers and computer architects alike. While several factors may contribute to the evaluation of how efficiently a circuit consumes energy, this thesis narrows down these factors to the following three pillars that will be explained and discussed in detail in the subsequent sections.

- **Computational Performance** This includes both the TPP (Theoretical Peak Performance) and the SP (Sustained Performance). The TPP is a metric that measures how fast a system can perform computations in an ideal setup where there is no overhead due to data movements for example. On the other hand, the SP measures how fast these computations are performed in a real scenario, including all possible overheads. For example, in high-performance computing, both TPP and SP are measured through the number of floating-point operations per second noted as FLOPS or FLOP/s.

In a software programmable many-core overlay, the term TPP refers to the peak computational performance of such an overlay. Here, the TPP is computed in a similar manner to the work in [51], by multiplying the amount of operations that can be performed by a PE in each clock cycle, the total amount of PEs and the operating clock frequency of the overlay. The corresponding equation that computes the TPP is given in **section** 3.3.

- **Energy and power efficiency** The power efficiency of a given computer system can be measured by dividing its SP by the corresponding consumed power and is expressed in FLOPS/W or FLOP/s/W. On the other hand, the energy efficiency is usually expressed in FLOP/Joule. The energy efficiency can in fact be deduced from the power efficiency because 1 W is equal to 1 Joule/s. Therefore, it is common for the energy efficiency to be used interchangeably with the power efficiency, because they mostly reflect the same intent. The power efficiency's equation will be given in **section** 3.3.

- **EPR** The Effective-to-peak Performance Ratio was used in [9] to indicate the ratio between actual performance and peak performance. This terminology is used here to indicate the ratio between SP and TPP in a software-programmable many-core overlay context which indicates the related computational efficiency (the detailed definition of SP and TPP is given in the beginning of this **section**, and their equations are given in **section** 3.3). This metric relates to how efficiently the real computation is performed and how close it comes to nearing the TPP. Nonetheless, this metric hides another important meaning in disguise. It can in fact, give an idea of the percentage amount of overhead that needs to be reduced to approximately achieve the best possible energy efficiency in a system. The EPR's equation will be given in **section** 3.3.

The work in [1], uses the term intrinsic energy to define the lower bound on the energy needed to perform a computation assuming an ideal system with optimal dedicated datapaths resulting in near zero energy overhead, implicitly hinting on a system operating at its maximal theoretical performance.

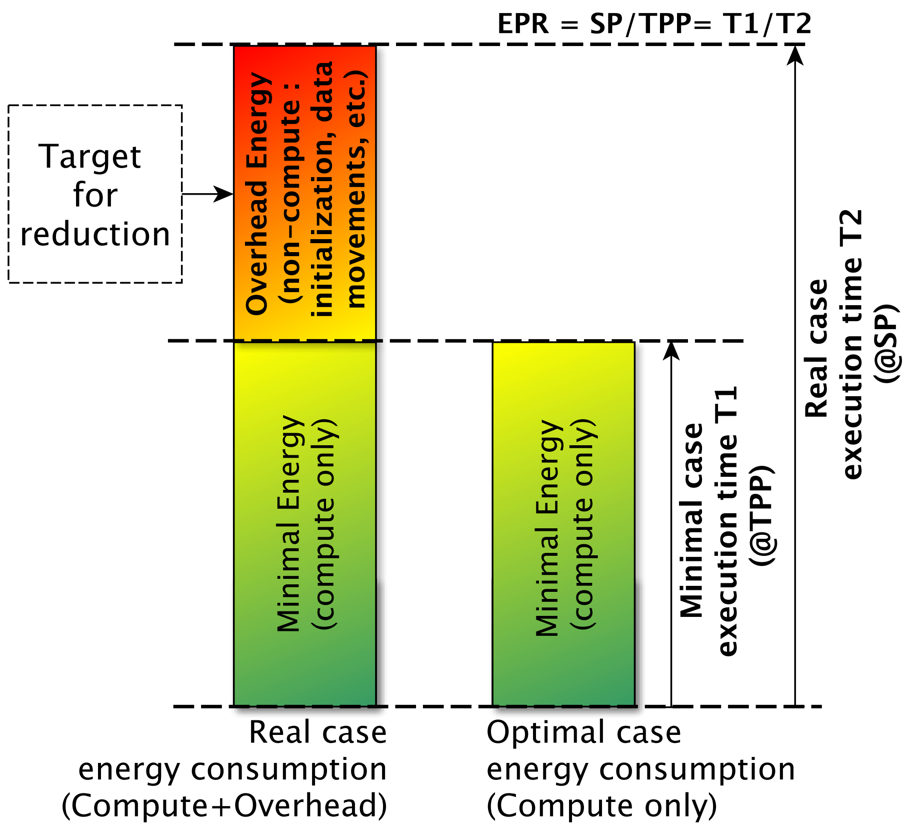

EPR = SP/TPP= T1/T2
Target for reduction
Overhead Energy (non-compute : initialization, data movements, etc.)
Minimal Energy (compute only)
Minimal Energy (compute only)
Minimal case execution time T1 (@TPP)
Real case execution time T2 (@SP)
Real case energy consumption (Compute+Overhead)
Optimal case energy consumption (Compute only)

Figure 3.1: Overview of the hidden impact of the EPR on energy overhead.

In an ideal system, the shortest execution time is obtained when the system achieves 100% of its TPP and is shown as `T1` in Fig. 3.1. The ideal minimal energy consumption is limited by `T1` as shown in Fig. 3.1 and is the closest match to the definition of intrinsic energy in [1].

In contrast, a real system exhibits multiple causes of energy overhead. Arguably, the most important reason is the energy consumed when moving the data to/from the computing blocks while not performing the desired computation in the mean time. In instruction-set-based systems, this data movement increases the number of instructions without increasing the number of performed operations, resulting in an increased execution time (reduced SP) and an increased energy consumption.

Therefore, when designing an instruction-set-based processing system, the EPR can add a valuable information that complements the power efficiency metric by showing to which extent the overhead energy has been reduced, by means of reducing the execution time, assuming the average overhead power remains constant. Mainly, this energy consumption reduction is a direct result of the execution time reduction that can be achieved by limiting the effect of data movements on energy through an efficient overlapping with the effective computations, at the instruction set architecture level. As a result, a higher EPR would implicitly hint at a more efficient use of the available energy.

3.2 Background

3.2.1 Power dissipation and energy consumption in FPGAs

In CMOS (Complementary metal–oxide–semiconductor) -based digital circuits, the overall power dissipation can be split into static and dynamic parts. The static power is the result of standby current flow in CMOS transistors and the leakage power due to the reverse-bias current of the formed diode in MOSFET (Metal–Oxide–Semiconductor Field-Effect Transistor) transistor's semiconductor junctions [1]. On the other hand, the dynamic power is the result of short circuit power and the capacitive load power. The short circuit power originates when complementary MOSFET transistors (n-type and p-type MOSFETs) in a CMOS circuit are briefly behaving as closed switches and pass a short circuit current during switching transitions [1]. On the other hand, the capacitive load power is the result of charge and discharge of circuit capacitance, due to clock toggling [1].

In general, the dynamic power dissipation accounts for the vast majority of the overall power in CMOS digital circuits which includes FPGAs, however, this remains depending on multiple factors, among which, the resource utilization percentage of the overall device or the operating clock frequency. The work in [89] estimates the dynamic power dissipation in a range of 85% to 90% of the overall power. While this estimation might be outdated (from the year 2000), more recent data still report a high percentage of dynamic power dissipation in FPGAs. For example, It has been reported (on the year 2015) that a Cyclone V SoC (System on Chip) [90], which consists of an FPGA and a HPS (Hard Processor System), dissipates 23% of its power in the HPS part (both dynamic and static) whereas the FPGA part dissipates the remaining as 69% dynamic power and 8% static power. Besides, the work in [91] reports dynamic power percentages nearing 90% when increasing the FPGA utilization. Nonetheless, the work in [89] provides a first order approximation of the dynamic power $\mathbf{P}_{dyn}$ shown in Eq. 3.1, where freq is the clock speed (expressed in Hz), $\mathbf{V}$ is the supply voltage of the circuit (expressed in Volt), $\mathbf{C}$ is the capacitance that is charged or discharged during a transition activity (expressed in Farad) and finally α is the the probability that such a transition occurs. Reducing the value of any component of this equation may reduce the power dissipation but not necessarily improve the energy efficiency. In fact, reducing the supply voltage increases signal delays which degrades the computational performance, similarly reducing freq decreases as well the computational performance which leads in both cases to an increase of the execution time and degrades the energy efficiency. Consequently, the most logical approach to decrease the dynamic power and hence the overall power dissipation would be to reduce the overall capacitance (in particular routing capacitance) as well as the probability that undesirable transitions occur [89].

$$\mathbf{P}_{dyn} = \alpha \times \mathbf{C} \times \mathbf{V}^2 \times freq \tag{3.1}$$

In this thesis, the term power dissipation refers to the average total power that is the sum of both static and dynamic power, and its value will be obtained by the reports generated by the FPGA vendor software analysis tools (Vitis and Vitis analyzer), that are used to profile the FPGA during run-time. These software tools allow obtaining highly accurate execution time and reports of power dissipation by inserting and monitoring extra profiling logic into the FPGA-based user-provided designs. The obtained execution time of a given application running on an FPGA, can be used later to compute the corresponding energy consumption which is simply the product of that time by the average power dissipation. The execution time is used as well to compute the energy efficiency or the power efficiency of the underlying FPGA circuit, as will be explained further in the experimental evaluation chapter.

3.2.2 A survey of techniques for energy-efficient FPGA-based design

The aim for energy efficient FPGA overlays faces many challenges, notably because it is not the only target to achieve. In fact, the computational performance is another equally important goal that should not be discarded. Often, a high computational performance would eventually increase the power efficiency and consequently the energy efficiency of a given design assuming it slightly increases or better maintains a constant average power dissipation. Nonetheless, the increase in the computational performance involves multiple factors, among which, raising the circuit clock speed, that can cause the power dissipation to be increased considerably. Consequently, empirical evaluation may be the best indicator of the effectiveness of any given energy-oriented design approaches. Besides improving computational performance, a designer may recur to low-power design techniques without compromising the computational performance.

- **Techniques proposed for digital CMOS circuits in general** Several works surveyed and summarized multiple guidelines for low-power design of digital CMOS circuits. For example, the work in [89] presented general design flow techniques to achieve this goal. It proposed three abstraction levels for power reduction, namely, system, architecture and technological levels. First, at the system level, the proposed techniques include system partitioning, scheduling and compression methods. Second, at the architecture level, these techniques include parallel hardware and hierarchical memories. Finally, at the technological level, it suggested reducing supply voltage, on-chip routing and control of the clock frequency. Furthermore, the work in [89] discussed other techniques that aim to reduce undesirable switching activity, such as clock control or the use of gray code counters, for example, for program counters among others, that are known to change the state of a single bit at each increment, which effectively minimizes transitions.

  The work in [92] particularly surveyed architecture-level power- and energy-efficient design techniques, among which, it reported instruction issue width optimization and pipeline balancing [93], clock gating [94, 95, 96], power gating [97], instruction queue resizing [98, 99, 100] and register file access optimization [101].

- **Techniques proposed specifically for reconfigurable chips** Some of the previously reported techniques, such as power gating are only applicable in full-custom chips such as ASICs, where a designer exercises a full control over all design aspects. A reconfigurable chip such as an FPGA is a semi-custom device where the physical resources are already available and unless this feature has been accounted for, it remains not possible to apply.

  Subsequently, the work in [102] presented some low-power-oriented techniques that are applicable in an FPGA context. Among these techniques, the reduction of dynamic power on clock scheme through clock gating at chip, design or RTL level, as well as the reduction of global resource power through the reduction of fanout and signal loads on global networks. Another interesting technique is to reduce the power dissipated by the implemented logic for example by removing glitches through balanced paths or through using gray code counters and optimized encoding of FSM (Finite State Machine) state transitions. The work in [102] also reported two other techniques targeting FPGA-based soft-processors, namely, specialized instruction-set extensions [103] as well as the recoding of instructions coupled with power-aware scheduling schemes [104].

Despite being an interesting feature, low-power is not always equal to energy-efficient. This is particularly true in instruction-set processors where faster processors may dissipate more power but perform computations in significantly shorter time, thus, consuming less energy and leading to a higher energy efficiency. Consequently, the work in this thesis, aims first at shortening execution time of software applications by the means of a high computational Sustained Performance (this implies a high EPR which results in a reduced energy overhead). Then, through micro-architecture enhancements, that aim at lowering the average power dissipation. Consequently, further details on the techniques that allow to achieve these goals will be discussed in later sections.

3.3 Important metrics for energy efficiency evaluation

- **Computational Performance [GFLOPS]:**

  The TPP represents the upper bound in computational performance of a given computer system. It defines the theoretical (ideal) performance limit of such a system, in which case all of its compute elements are fully used to perform useful computations at every clock cycle, while new data are always available at every input operand.

  In a programmable many-core overlay architecture where multiple processing elements operate concurrently, the TPP can be expressed as Eq. 3.2, where Freq is the operating clock frequency in Hz, $\mathrm{N_{PE}}$ the number of PEs and $\mathrm{N_{FLOPs}}$ is the amount of floating-point operations that can be executed in a single PE at every clock cycle.

$$\mathrm{TPP\ [FLOPS]} = \mathrm{N_{PE}} \times \mathrm{N_{FLOPs}\ [FLOPs]} \times \mathrm{Freq\ [Hz]} \tag{3.2}$$

  For example, assuming that this programmable many-core overlay operates at 200 MHz

(Freq [Hz] = 200 x 10^6) and contains four PEs ($N_{PE} = 4$) where each PE embeds a floating-point multiply and add unit that is capable of performing one addition plus one multiplication ($N_{FLOPs} = 2$) at every tick of the clock signal, based on Eq. 3.2, the TPP can be evaluated as in Eq. 3.3.

$$\text{TPP [FLOPS]} = 4 \text{ [PEs]} \times 2 \text{ [FLOPs]} \times 200 \text{ [MHz]} = 1.6 \text{ [GFLOPS]} \tag{3.3}$$

In practice, the SP, that is the real computational performance, is dependent on the target application as well as the computer system architecture. One way to compute the SP is to first measure the wall clock time ($T_{\text{wall\_clock\_time}}$) from the start until the end of a program execution, then manually compute the number of useful computations performed during this time (#FLOPs). Consequently, SP can be obtained through Eq. 3.4.

$$\text{SP [FLOPS]} = \frac{\text{\#FLOPs [FLOPs]}}{T_{wall\_clock\_time} \text{ [s]}} \tag{3.4}$$

Often, since the SP is relative to the underlying application, the TPP is used instead as a measure of the computational performance of computer devices, whereas the SP is used when computing the power-efficiency under a specific workload.

- **Power Efficiency [GFLOPS/W]:**

  The Power Efficiency as its name suggests indicates how well a computer system consumes power under a specific computational workload. It is expressed as the ratio of sustained (real) performance to the average power consumed by the underlying computation. As such, it can be expressed by Eq. 3.5 where POWER is expressed in Watt (W). In FPGA devices, the value of POWER for a design can be captured during run-time by enabling the related power profiling switches using software tools such as Vitis (for Xilinx FPGA).

$$\text{Power Efficiency [FLOPS/W]} = \frac{\text{SP [FLOPS]}}{\text{POWER [W]}} \tag{3.5}$$

The energy efficiency is often used interchangeably with the term power efficiency to express the same intent. The energy efficiency is typically expressed in [FLOPs]/[Joule]

(FLoating Point Operations per Joule) and can be easily derived from the power efficiency knowing that 1 [Watt] = 1 [Joule/second]. In other words, [FLOPs]/[Joule] is similar to [FLOPS]/[W].

- **EPR** [%]:

  The Effective-to-peak Performance Ratio can be obtained through Eq. 3.6.

$$\text{EPR [\%]} = \frac{\text{SP [FLOPS]}}{\text{TPP [FLOPS]}} \tag{3.6}$$

  The concept of the EPR is illustrated in Fig. 3.1. The EPR can be a good indicator of how optimized is the computer architecture but also it can implicitly hint on how optimally the related energy is being used. At the theoretical peak performance, the execution time is minimal and so is the related consumed energy, assuming a constant average power dissipation. Thus, by maximizing the sustained performance to an extent close to the theoretical peak, the overhead energy is reduced by minimizing the total execution time which leads to a more efficient use of energy. For example, an EPR of 90% indicates that 90% of the execution time had been dedicated solely to useful computations, whereas only the remaining 10% of that time had been spent on non-compute operations, such as initialization, data movements between processing elements and memory loads or stores.

3.4 Levels of energy-efficiency improvement

Energy-oriented design improvements heavily impact the energy efficiency of FPGA overlays. The related impact degree varies depending on the level at which these improvements were introduced. In general, this degree tends to be more visible when the abstraction level is higher. Based on the reference [2], the work in [1], showed these abstraction levels and the general extent at which it can be possible to reduce circuit power as can be seen in Fig. 3.2.

In digital circuits such as ASICs (Application-Specific Integrated Circuits), energy-oriented design techniques can be applied at all these levels thanks to the full-custom nature of these circuits where a designer intervenes in all the product life-cycle processes. In contrast, reconfigurable chips such as FPGAs are semi-custom devices, where the circuit is pre-fabricated and only a

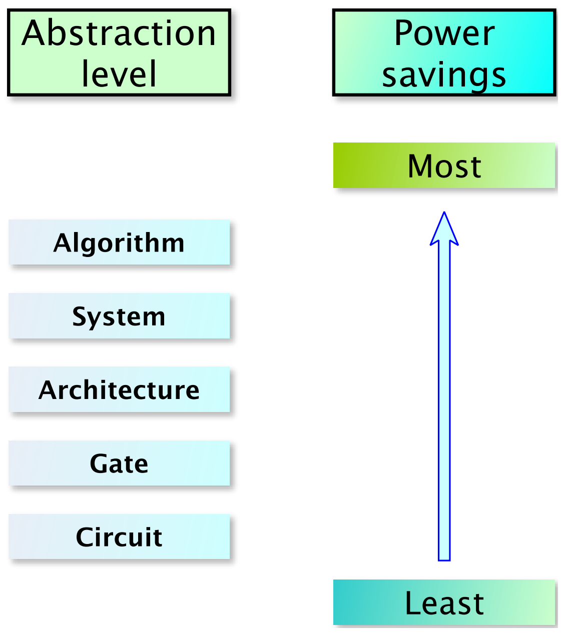


Figure 3.2: The impact of design abstraction levels on power reduction [1],[2].

limited set of changes can be applied in post-layout, through the process of reconfiguration. As such, ASIC-only energy-saving techniques such as power-gating, among others, are not applicable to these devices. Worse yet, maintaining a high-degree of application flexibility through an overlay-based abstraction of FPGAs, sacrifices some other possible paths for saving energy, such as when inferring logic through high-level HDL-based circuit descriptions, instead of manually instantiating FPGA blocks like DSPs which prohibits the use of power-saving switching logic inside of these blocks, such as the one disabling the use of internal multipliers when not required. In this section, the two levels of energy-efficiency improvements in the proposed many-core overlay architecture, are presented and discussed. The first is the ISA (Instruction Set Architecture) level which provides the hardware/software interface, and the second is the micro-architecture which concerns the RTL-based description and the corresponding physical implementation.

3.4.1 Instruction Set Architecture

- **Optimal multi-aspect encoding of instructions** ISA design is a complicated task that involves in-depth computer architecture knowledge and expertise. The encoding of instructions fields, such as their width and relative placements inside the instruction plays an important role in the ease of use and the ease of micro-architecture implementation of an ISA. It may also have an impact on the power outcome of such an implementation. For example, preserving the position of a certain field across different types of instructions, simplifies the decoding logic and reduces its related circuit size. Furthermore, it can reduce issues related to critical paths for signals with a high fanout such as the immediate fields, across different implementations [105].

Besides, most available commercial or open-source ISAs are designed in a device agnostic way. That is, these ISAs are not designed to address a specific device with the pre-assumption of constrained resources. Hence, prior to the design of energy-efficient ISAs, that are targeting FPGA devices, it is important to study the proprieties of their building blocks, such as their on-chip memories. In particular, these memories exhibit similar power consumption trends, whereas their storage spaces are fully or partially used. Hence, it makes sense to fully use their storage capacity to the fullest extent possible without compromising other aspects of the overall ISA design such as the full width of the instruction which should be minimized to reduce programs memory footprint. Xilinx FPGAs provide two kinds of the largest capacities of on-chip memories in FPGA devices called BRAMs and URAMs. These two kind of memories can hold up to 512 (2^9) and 4096 (2^{12}) 64-bit data, respectively. They also can be used to implement register files and local memories in a processing element design. As such, using a direct addressing mode, it has been decided to implement the DRAGON ISA with a 12-bit-width field for local and broadcast memory addresses and 8-bit-width fields for the Register File (a 9-bit-width field would have increased the instruction width to more than 64-bit width which complicates the storage of instructions).

Furthermore, instruction-set based FPGA-overlays are mostly used to address limited set of applications, therefore, it is advised to keep a limited number of base ISA instruction opcodes, while providing a way to ISA extensions, to increase application flexibility through the possibility of more complex operations. Consequently, the limited number of opcodes reduces the decoding and execution resources which positively impacts the energy efficiency across different circuit implementations.

- **Efficient ISA-level overlapping of computation and data movements**

To maintain the energy consumption near its ideal minimal level shown in Fig. 3.1, it is

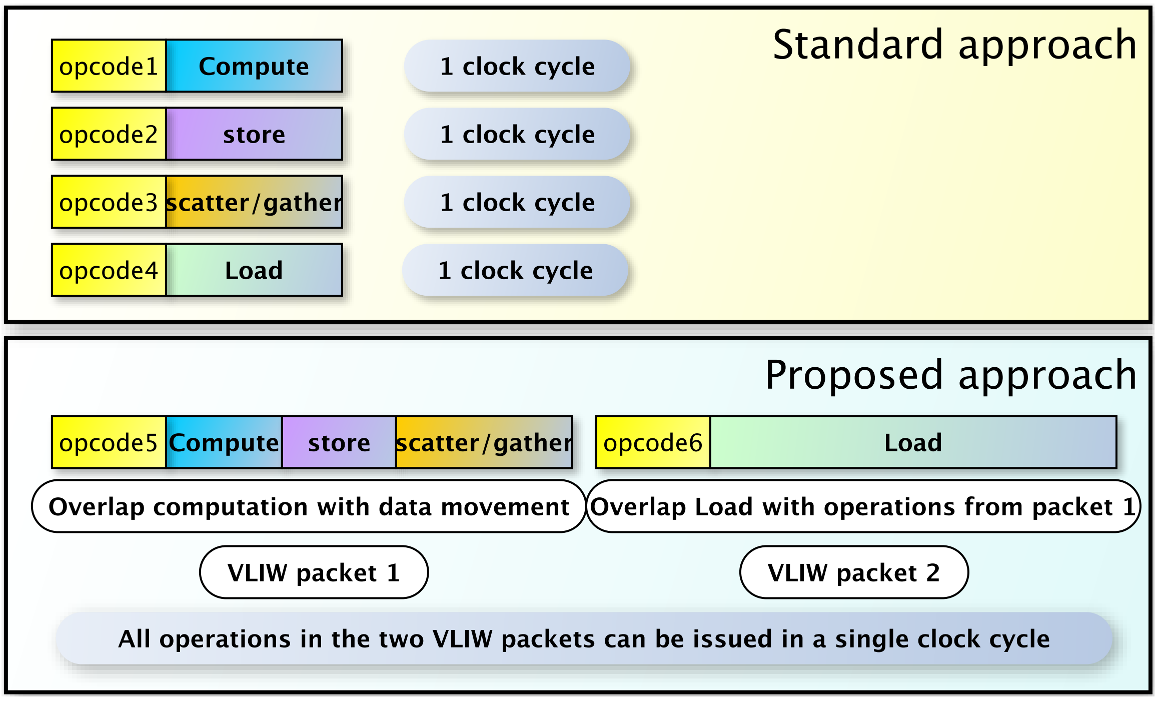


Figure 3.3: Overlapping of compute and data movement operations using a single clock cycle instruction issue.

necessary to minimize the application execution time. The processing time of a computation depends upon the operating clock frequency, the number of issued instructions per clock cycle and the total number of instructions required. While the increase in frequency improves the computational performance, it may inadvertently increase the dynamic power consumption due to the increased switching activity in CMOS transistors. The impact of increasing the clock speed on energy consumption is a matter of empirical experiments and can not be easily identified analytically. Therefore, to minimize the execution time the focus should be mostly on reducing the overhead energy by the means of minimizing the overall number of computation clock cycles. This can be achieved by the parallel issue of concurrently executed instructions through a custom VLIW (Very Large Instruction Word) approach [11]. The other option would be to allow single instructions to perform multiple concurrent operations. A possible scenario would consist of fusing computations with data movement operations, resulting in a significant reduction of the overall number of instructions that leads to considerable gain in energy efficiency. Furthermore, to avoid spending more clock cycles waiting for new data to be available into the register file, the common VLIW approach can allow simultaneous Load operations, through a separate instruction in the second VLIW packet, together with the ongoing compute and store operations in the first VLIW packet instruction. This first packet may also allow the concurrent scattering of computed results towards neighboring processing elements as well as collecting their output data into local dedicated communication buffers. This approach also dramatically reduces the execution time by means of direct neighbor-to-neighbor data transfers, instead of the lengthy cycles using multi-memory-level transfers. These concepts are illustrated in Fig. 3.3.

- **Efficient memory architecture** Data movement is a major bottleneck for performance and energy efficiency. While several computing models, such as systolic arrays and data flow architectures, managed to alleviate its impact, it remains one of the hardest computer architecture problems in instruction-set processor design. In this context, an efficient memory architecture would reduce the negative impact of moving data between the global memory where it originally resides and the compute blocks where the useful work (computation) is performed. The work in [1] lists double buffering among the techniques that allow a smooth transfer of data in a pipelined manner to hide undesirable latencies that would otherwise decrease the computational performance and increase the execution time which leads to an increased loss of energy. The EXACC architecture [8] implements an efficient memory architecture consisting of three levels (Global, Broadcast, Local) where the intermediate level (Broadcast) acts as a mean of double buffering. The proposed DRAGON architecture builds upon this memory model and provides ISA support (through a special LDBM instruction) allowing the intermediate Broadcast memories to act as an intermediate DMA (Direct Memory Access) engine to move a whole chunk of data from BMs (Broadcast Memories) to LMs (Local Memories) in a single instruction, which allows the PEs to perform useful computations on existing data stored on the RegisterFile for example, while new data is being loaded into their LMs.

- **Multi-source operands** As opposed to the CISC (Complex Instruction Set Computer) architectures, a RISC (Reduced Instruction Set Computer) architecture prohibits direct operations on memory operands and moves the data from memory to the Register File prior to operation execution. In the context of many-core processors, this increases the overall energy consumption by introducing undesirable latencies. The proposed DRAGON ISA alleviates this issue by adopting an aggressive model that lies in between CISC and RISC, that allows direct operations on communication buffers used as a mean to temporarily store exchanged data between neighboring PEs, in a manner bypassing the multi-memory-level transfer of data that can be found in several many-core-based processing architectures such as GPUs [106].

3.4.2 Micro-architecture and physical implementation

- **Spatial parallel execution** The execution time is an important factor for energy consumption of computer systems. In fact, reducing the computation time while maintaining the same average power consumption would lead to an improved energy efficiency. Among the most effective execution time reduction techniques is the parallel processing of data. In modern day computing, the most used forms of parallel processing from Flynn's taxonomy [107] are SIMD and MIMD (Multiple Instruction Multiple Data). In particular, the first is known to be the more energy efficient option as it will consume less power because of the shared instruction stream that shrinks the memory requirements to store program instructions and also because of the significantly simpler control scheme that removes redundant logic [12].

- **DSP Utilization**: Modern large FPGAs such as the one in [37] offer a relatively large amount of DSP blocks [63]. These resources are capable of performing several kinds of computations including addition, multiplication and bit-wise operations while using a tiny fraction of the overall system power. Oftentimes, the implementation tools automatically map the described circuit behavior to these resources to minimize the design size and improve its performance. Nonetheless, poor HDL-based circuit description can instruct the tool to use other resources such as LUTs (LookUp Tables) which complicates routing, enlarges the design size, and increases its power consumption. Therefore, it is necessary to understand and follow the vendors guidelines for automatically inferring DSP resources to implement a desired functionality. This implies proper HDL coding that allows the software tool to easily capture the intended circuit behavior and successfully map it to

DSP blocks instead of LUTs. It is also possible to force the tool (at the possible extents) to utilize these DSP blocks for certain functions by adding a special pragma (* use_dsp = "yes" *) at the RTL level. This is especially useful when the granularity of the computation is considerably large, such as in double-precision floating-point computations.

- **Fanout Reduction** In a digital circuit, it is common for a single output source signal of one logic block to be connected to multiple destination inputs of other logic blocks. For example, in the case of a SIMD architecture, a single instruction stream signal has to be connected to several processing elements. This leads to the Fanout concept, defined by the number of terminal destinations that are fed by a single source signal. In FPGA devices, there are typically two types of routing resources, global and non-global. The global ones are implemented in a well defined tree-structure that shortens the overall delays and therefore they are usually dedicated to clock and reset signals usage. On the other hand, the non-global ones are usually used by all other signals. Depending on the FPGA structure, there are thresholds where a high-fanout signal may create congestion and complicate routing resulting in poor quality of results, reduced clock speed and increased power consumption due to the increased capacitive load of the resulting long connection wires. While it may be possible to use global route resources for high-fanout non-clock and non-reset signals, it is usually not recommended or restricted to a minimum threshold because it would adversely lead to higher delays. It is common for routing tools to automatically solve this kind of issues by promoting a high-fanout signal to a global signal that uses global routing resources or by simply replicating the gate that drives this signal. However, the outcome result is not guaranteed, especially for large designs that can easily suffer from congestion. Therefore, it is more efficient to address this problem manually at the RTL level, by manually replicating high-fanout signals or by guiding the tool through specific pragmas.

- **Pipelining** Multi-level combinatorial logic can lead to unbalanced delays between the different signals arriving at the inputs of a destination logic gate. These undesired delays may unnecessarily introduce a power-consuming switching activity, that is called a glitch [108]. The work in [109] reports, based on [110], that this kind of issue can be responsible for up to 70% of the overall power consumption in ASIC circuits. The work in [109] also reports that this percentage can be easily surpassed in FPGA devices and shows an example 32x32 non-pipelined multiplier where the consumed power due to glitches accounts for about 96.9% of the overall power consumption.

In a digital circuit, the critical path, that is the path with the longest delay between two registers dictates the operating clock speed of the full circuit. To improve this speed, designers often recur to a well-known technique called pipelining, which consists of splitting the critical path further, by inserting intermediate registers, in order to reduce the logical levels between both ends of the original path. The work in [109] suggests that pipelining can be used to reduce the occurrence of glitches and hence the related power consumption. In fact, reducing the amount of logic between two registers can effectively reduce the spurious switching probability in a given circuit. The study in [109] backs its claim by showing the impact of pipelining on the dynamic glitching power in a multiplier circuit. As a result, it shows that an 8-stage 32x32 pipeline multiplier can cut the dynamic glitch power by almost 50%. Nonetheless, while pipelining reduces the glitching activity, it increases the number of functional transitions. The work in [109] suggests that to effectively reduce the overall dynamic power, it is necessary that the reduction in glitch transitions out-weights the increase in valid transitions which is the case for large designs including their example of the 32x32 multiplier.

In many-core instruction-set processor designs, there are several opportunities to apply pipelining where the benefits can lead to reducing the dynamic power consumption due to glitches but also to improving the circuit performance. For example, a pipelined processing element can benefit from temporal parallelism where multiple separate instructions can be partially executed in the same clock cycle through different portions of the pipeline stages. Furthermore, pipelining can provide more flexibility to the FPGA synthesis tool to introduce retiming, which is a technique that consists of balancing signal path delays by moving registers forward or backward in the related path. Another scenario would be to pipeline large width data buses, in particular, those connected to off-chip or on-chip memories, in order to simplify the routing task, achieve better operating clock speed and reduce the dynamic power resulting mainly from glitches.

- **Design considerations for HBM-enabled multi-die FPGAs** Modern FPGAs are the sum of small to middle-sized dies stacked into a single chip by means of SSI (Stacked Silicon Interconnect) technology. These dies are called SLRs (Super Logic Regions) in Xilinx FPGAs and the wires between them called SLLs (Super Long Lines) [111] are scarce and consume more power than those inside the regions because of their size. Interconnects are responsible for an important part of the power dissipation in FPGAs. The work in [102] reports that they dissipate at least 34% and 60% of leakage and dynamic power, respectively. Therefore, it is recommended to avoid crossing the SLR regions or reduce the use of SLLs at the extents possible. In particular, for many-core overlay designs with a 3D interconnect degree and a balanced PEs distribution across SLR regions, it is possible to achieve this goal by keeping the interconnect wires for the 3rd dimension inside a single region by the means of a properly designed modular implementation and its related layout.

  While a balanced PEs distribution across regions leads to efficient resource utilization and enhanced scalability, it is offset by the fact that many HBM-enabled FPGAs implement their HBM memory banks solely in the bottom region. This complicates the process of connecting these memories to their end-buffers that may be placed into different dies and imposes the use of SLLs as a means of physical data transport. As such, it is necessary to analyze the relationship between a balanced logic distribution across regions and the efficient bandwidth utilization, to strike the best deals in terms of scalability, computational performance and power efficiency.

- **Manual floorplan** The architecture definition of a given circuit provides a clear view on the proximity of functional blocks (such as PEs) to each others, as well as the relative length of wires that interconnect them. Meanwhile, the synthesis tools may alter the architectural specification view without modifying its intended functionality, by combining or splitting design blocks. Worse yet, the implementation tools may put relatively close blocks from the architectural point of view, far apart in the physical placement on an FPGA. This leads to a serious problem. In fact, it complicates the routing process which results in larger delays, degraded performance and increased power dissipation. To solve this issue, it is highly recommended that the designer provides placement constraints to assist the implementation tools during placement step. This ensures maintaining a relatively close circuit view to the previously defined abstract architecture view. As such, a designer may define physical area boundaries on the FPGA to host parts of the design elements. A typical example would be to use what is called Pblocks in Xilinx FPGAs. Moreover, it is possible to use vendor pre-defined Pblocks, in the context of multi-die FPGAs, to pass the desirable placement constraints information during the pre optimization step, right after synthesis, to instruct the tools to map a large portion of a design into a specific chip die.

- **Low memory footprint for inter-PE data exchange** In the context of a tightly-connected many-core architecture, increasing the amount of adjacent PEs can significantly increase the number of communication buffers that are used to efficiently exchange data between these neighbouring PEs. These buffers consume energy and may limit the scalability of the overall system due to the lack of sufficient on-chip memory resources. Consequently, an efficient utilization of such kind of resources should aim at reducing the overall memory footprint without compromising the computational performance, thus, leading to an improved energy efficiency by means of area and power minimization.

3.5 Summary

This chapter introduced several concepts that will guide the design of the many-core overlay architecture proposed in this thesis. This chapter defines the common metrics used to evaluate the computational performance, the power (and energy) efficiency as well as the computational efficiency (EPR). Moreover, several state-of-the-art techniques that can be applied to improve the energy efficiency were surveyed and reported. Ultimately, specific techniques that were applied in this work have been presented and explained in detail.

Part II

Software Part: Accelerator softwarization

Chapter 4

The DRAGON Instruction Set Architecture

4.1 Energy-efficiency considerations for Instruction Set Architecture Design

The design of an ISA generally imposes several trade-offs in terms of flexibility, performance, code density, specialization, and energy efficiency. Often times, these trade-offs are mostly addressed in the micro-architecture (implementation) level because most of ISAs have been proposed decades ago and have been since frozen to allow compatibility with their large body of software applications. While most of the existing ISAs are commercial and may require a paid license and/or royalties, such in the case of the ARM ISA, RISC-V has been proposed as the Linux of hardware because it is possible to use it in an open source manner. Despite the attractive features of RISC-V, such as its modularity and the possibility to extend its base instructions, it may be better suited for ASIC implementation rather than reconfigurable chips, in particular from an energy efficiency perspective. In fact, an ASIC implementation has full control over all aspects of power optimizations, including at the physical implementation of cells and manufacturing technology process. In contrast, an FPGA provides access to a subset of what an ASIC can manipulate and as such energy-aware optimizations can only be accessed in higher levels of abstraction. Consequently, most considerations would be addressed in the architecture and micro-architecture levels and would primarily address optimizations of the dynamic energy consumption rather than the static one. Besides, the RISC-V ISA adopts a RISC architecture that is capable of executing computational instructions whose operands are incoming solely from the Register File which may limit the overall performance outcome. As explained in previous chapter, it is important to maximize the sustained performance close to its theoretical peak to minimize energy consumption overhead. While it may be possible to achieve this goal using the RISC-V ISA, it would still require heavy micro-architecture design efforts (for example an Out Of Order implementation with macro-operation fusing) that would significantly increase the resource utilization and power consumption without the proportional increase in performance. For these reasons, it has been decided to implement the DRAGON ISA from a clean slate, in a way that would be better suited to achieve performance and energy efficiency goals on reconfigurable chips such as FPGAs.

4.2 The Memory Architecture

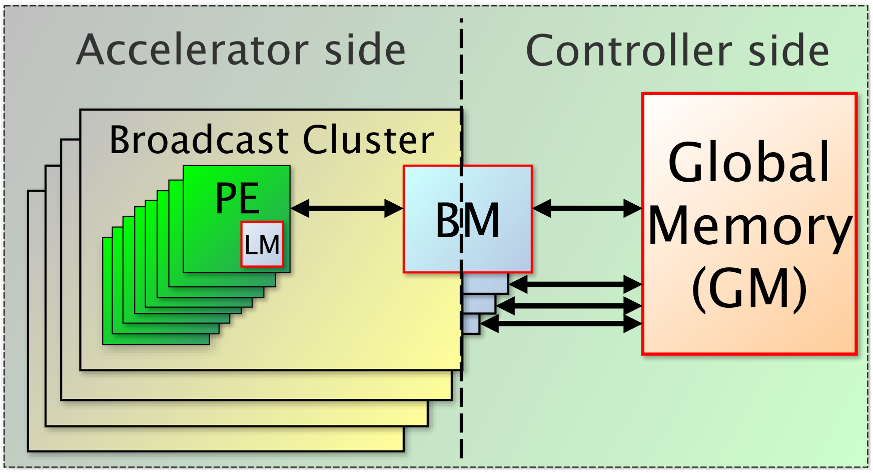


Figure 4.1: The DRAGON Memory architecture.

The proposed DRAGON ISA adopts a memory architecture model that consists of three levels, namely, in ascending order: The LM (Local Memory), The BM (Broadcast Memory) and on the last level the GM (Global Memory). Fig. 4.1 depicts these levels and their interactions. In fact, the GM has the largest storage that holds both program instructions and data, while BM interfaces between LM and GM to decouple data movements from execution. The GM is a 64-bit addressable memory, while LM is 12-bit addressable. The BM is implemented in a banked manner. A single BM bank is 12-bit addressable and each LM memory can write its data to a single BM bank. However, in a broadcast cluster, the whole BM banks address space can be accessed by all PEs and therefore by all LMs, through a broadcasting feature which allows directing a specific BM bank data to all the PEs of the same broadcast cluster.

4.3 Instruction Set Organization

In this section, the different categories of instructions and their general structure is depicted and explained. Some instructions that belong to the same category may slightly differ in structure. In other words, some bit fields may be used by some instructions and not others, despite belonging to the same category. The partial text in these bit fields may contain "|**un**" which means can be unused, depending on the instruction. Bit fields that contain only **un** are currently unused, regardless of the instruction, but can be reserved for future extensions.

NOP

63 58	57 0
0x0	0x0

The **NOP** instruction stands for "No Operation" and is a special one that do not belong to any particular category. It can be used as a filling packet in a VLIW implementation where one or all of the instruction packets should perform no operation. It simply advances the execution time by one clock cycle without impacting the logic state of the processing system.

4.3.1 General R-Type instructions

63 58	57 50	49 48	47 36	35 32	31 20	19 12	11 4	3 0
opcode	src1	md	lmaddr|un	off|un	bmaddr|un	src2|un	rdst|ndst|un	opsrc

The general **R-Type** instruction category performs an operation on two operands and outputs the results according to the instruction fields **md** (mode) and **opsrc** (operand source). The first operand always comes from a location in the register file which has 256 different locations and is addressed by the instruction field **src1**. Depending on the **opsrc** instruction field, the second operand can come from a location in the register file addressed by the **src2** instruction field, from a location in the communication buffers (up to eight different buffers corresponding to eight different PE neighbors can be selected through **opsrc**), or from a location in the broadcast memory addressed by **bmaddr** (Broadcast Memory Address).When **opsrc** is set to 0xF, the **off** (broadcast offset) instruction field, selects the BM bank from which the data located in the corresponding **bmaddr** location should be broadcasted to all PEs in the Broadcast Cluster. Otherwise, each BM bank data is broadcasted only to its corresponding PE. In total, in a broadcast cluster there are exactly 16 PEs and 16 BM banks and every PE can be connected to up to 8 PE neighbors.

When the **md** field is set to 0b00 the result of the operation is stored back to the register file in its location addressed by **rdst** (register file destination). When the **md** field is set to 0b01 the result of the operation is stored into the local memory addressed by the **lmaddr** field. Otherwise, when the **md** is set to either 0b10 or 0b11, the operation result is scattered towards PE neighbors and stored into their corresponding communication buffer. In this case, the **rdst** field becomes the **ndst** (Neighbor Destination) which sets the direction of data scattering.

Table 4.1 and Table 4.2 summarize the different possible combinations with each **R-Type** opcode. Besides, by using the pseudo-instruction extension mnemonics in these tables, it is possible to define 48 different operation for each base **R-Type** instruction, in the form of (without the parenthesis):

(**opcode**-mnemonic).(**opsrc**-extension-mnemonic).(**mode**-extension-mnemonic)

For example, an **ADD** instruction can have 48 pseudo-instructions. When the the **opsrc** is set to 0b1111 and the **md** field is set to 0b00, the resulting pseudo-instruction becomes : **ADD.BMBR.SRF**.

In the base DRAGON instruction set there are 13 different R-Type base instructions. Given that there are 48 different combination per base R-Type instruction, a total of 624 different R-Type operations can be performed.

In general, the R-Type instructions are able to perform operations using the ALU (Arithmetic and Logic Unit) or MAC FPU (Multiply-ACcumulate Floating-Point Unit), with two operands. The first operand always comes from the output of the RegisterFile, selected by the instruction field **src1**, while the second operand can have multiple input sources. These sources can be either an immediate value, provided by the **imm** instruction field, the second output of the RegisterFile

mode (md)	**behavior**	**pseudo-instruction extension mnemonic**
0b00	Store result into register file	SRF
0b01	Store result into LM	SLM
0b10	Scatter result to neighbor PEs	SNPE
0b11	Store result into LM and scatter to neighbor PEs	SNPE

Table 4.1: The **mode** instruction field behavior and mnemonics for pseudo-instructions.

opsrc	**second input operand**		**pseudo-instruction**
	source	**location**	**extension mnemonic**
0b0000	register file	RF[src2]	RF
0b0001	immediate value (from two VLIW slots)	-	imm{imm1[63:48],imm2[47:0]}
0b0010	Broadcast Memory	BM[bmaddr]	BM
0b1111	Broadcast Memory	BM[off][bmaddr]	BMBR
0b0011	North Neighbor Buffer	N-FIFO output	NNB
0b0100	West Neighbor Buffer	W-FIFO output	WNB
0b0101	East Neighbor Buffer	E-FIFO output	ENB
0b0110	South Neighbor Buffer	S-FIFO output	SNB
0b0111	Remote North Neighbor Buffer	RN-FIFO output	RNNB
0b1000	Remote West Neighbor Buffer	RW-FIFO output	RWNB
0b1001	Remote East Neighbor Buffer	RE-FIFO output	RENB
0b1010	Remote South Neighbor Buffer	RS-FIFO output	RSNB

Table 4.2: The source input operands with each **opsrc** instruction field entry.

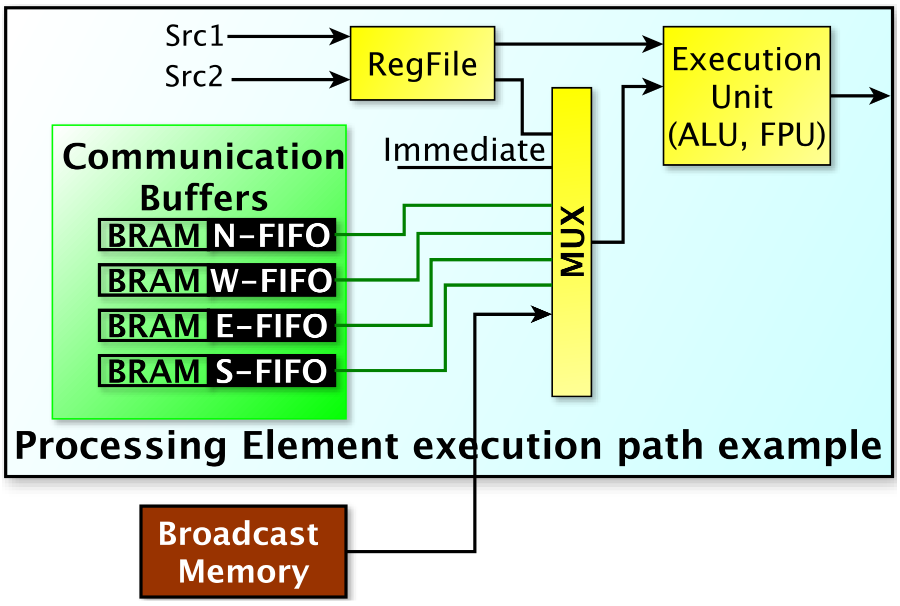
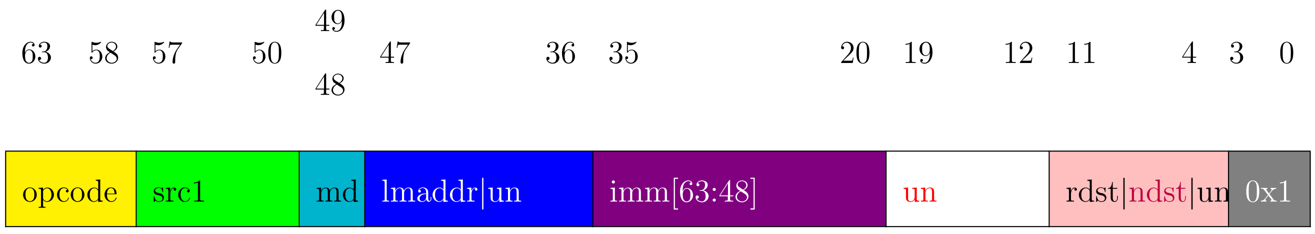


Figure 4.2: Possible source operands for R-Type instruction execution.

that is selected by the **src2** instruction field, the data provided from one of the broadcast memory banks as well as any of the data stored into the communication buffers. Fig. 4.2 depicts the possible operand inputs for the case of a 2D interconnect, embedding four communication buffers. A higher interconnect degree would increase the number of communication buffers sources. The baseline ISA supports data exchange with up to eight neighboring PEs.

4.3.2 Immediate R-Type instructions (Pseudo-instructions)



When the **opsrc** instruction field is set to 0x1, the second operand of an R-Type instruction becomes an immediate operand input whose explicit value is stored using both VLIW packets. The first packet contains the Upper 16 bits of the 64-bit immediate value in instruction[35:20], effectively replacing the **off** and **bmaddr** instruction fields. The lower 48 bits of the 64-bit immediate value reside in instruction[47:0] of the second 64-bit packet of the 128-bit VLIW instruction.

It is worth noting that the instructions involving immediate operations in the R-type format do not have separate opcodes. In fact, in order to reduce the instruction count and simplify overall operation and implementation, an architectural choice was made for not deploying any new opcodes for the register-immediate operations (where first operand comes from register file while second operand is an immediate value) and instead using these special instructions by fixing the **opsrc** bit field to 0x1 and possibly using pseudo-instructions.

Because they are destined to the Execution Slot of the VLIW PE, **R-Type** instructions should be packed only into the first VLIW instruction packet slot.

4.3.3 C-Type instructions

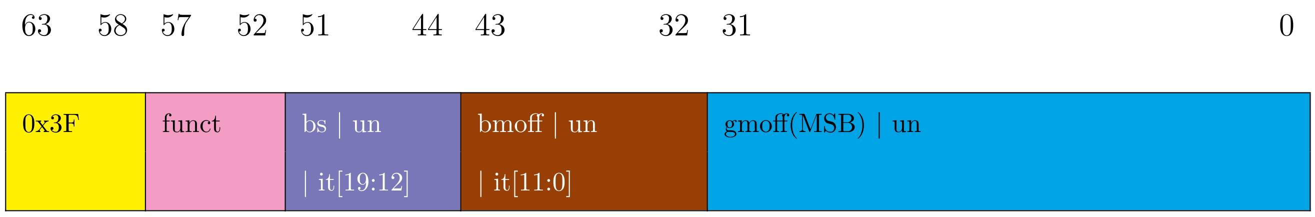
The **C-Type** instructions are control instructions that are executed on the controller part with impacts that may reach the accelerator part (the controller-side of the broadcast memories) through **RDGMEM** and **WRGMEM** instructions. The **C-Type** regroups just five base instructions, namely, **REPEAT**, **BNZ**, **RDGMEM**, **WRGMEM**, and **STOP** instructions. These instructions have a fixed opcode equal to 0b111111 and use the **funct** instruction field instead to distinguish each operation.

The **REPEAT** instruction performs a loop for a number of iterations specified by the instruction bit fileds it[19:12] and it[11:0] which are concatenated to create a 20-bit iterations value.The **REPEAT** instruction should be followed by the instructions inside the desired loop and then enclosed by the **BNZ** instruction, which branches to the loop body when the decremented number of iterations is different than zero. Otherwise, the **BNZ** exit the loop body and the normal operation of the program continues outside the loop.

The **RDGMEM** instruction allows moving a block of data from the Global Memory to the Broadcast Memories through the AXI protocol. The **bs** (burst size) instruction field instructs the desired number of data bursts to be moved. DRAGON is tightly connected to hardware implementations and therefore it assumes the programmer knows the AXI bus width that connects the DMA to the Broadcast Memories. On the Accelerator side, the data is written to the Broadcast Memories using the offset marked by **bmoff** (bmoffset) bit field.

The **WRGMEM** instruction uses the **bs** (burst size) and **bmoff** bit fields in the same way explained for the **RDGMEM** instruction. However, The **WRGMEM** is used in the opposite direction of data transfer. That is a **WRGMEM** instruction moves data from Broadcast Memories to the Global Memory.

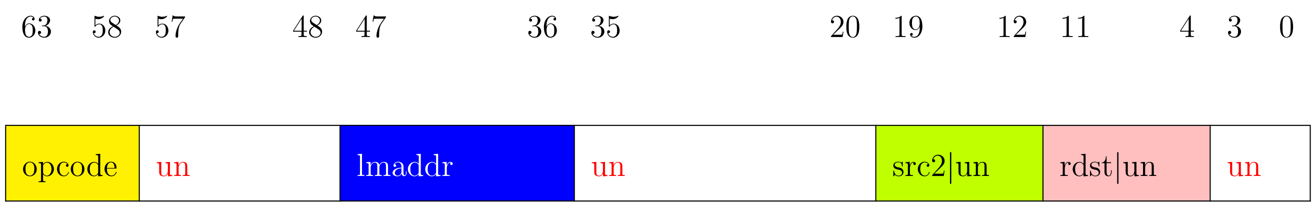
The GM is addressed by 64-bit address pointers. Therefore, both the **RDGMEM** and the **WRGMEM** instructions occupy the full dual-packet VLIW slot because they use direct addressing mode and specify the GM offset directly on the instruction. The first packet contains the upper 32-bit GM offset marked by the **gmoff** bit field, whereas the second VLIW packet contains the lower 32-bit GM offset on its lower 32 bits.



4.3.4 LM-Type instructions

LM-Type instructions regroup local memory loads (**LD** opcode) and stores (**ST** opcode). A **LD** instruction loads data from the address location **lmaddr** of LM to the register file destination address **rdst**.In addition, with the exception of **rdst** field, all instruction fields marked as **un**

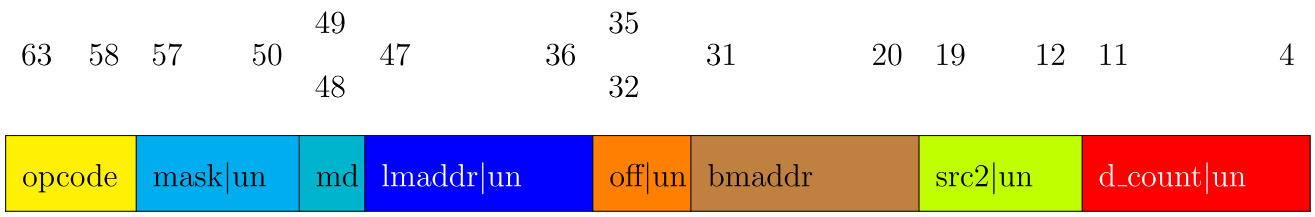
are unused bit fields.



In the case of a **ST** instruction, the 64-bit data stored into the register file address location marked by the instruction field **src2** is stored into the local memory address location marked by the bit field **lmaddr**. In addition, with the exception of **src2** field, all instruction fields marked as **un** are unused bit fields.

Because they are destined to the Memory Slot of the VLIW PE, **LM-Type** instructions should be packed only into the second VLIW instruction packet slot.

4.3.5 BM-Type instructions



BM-Type instructions regroup broadcast memory loads (**LDBM** opcode) and stores (**STBM** opcode). The data stored into BM using the **STBM** instruction, may be incoming either from an address location inside the local memory or the Register File, depending on the **md** instruction bit field, as shown in Table 4.3. The first is addressed through the **lmaddress** instruction bit field, the second through **src2** instruction bit field. The target address location into BM is marked by the **bmaddr** instruction bit field. The remaining bit fields containing **un** are unused by the **STBM** instruction.

On the other hand, the **LDBM** instruction can provide a DMA-like behavior by transferring a burst of data to the LM, through a single instruction call. The **d_count** (data count) field of the instruction, dictates the amount of the data to be copied from BM to LM. The **lmaddr** and **bmaddr** instruction bit fields set the base addresses of LM and BM, respectively. The data can be broadcasted from a specific BM bank to all PEs by setting the lower bit of **md** (mode) to 0b1 and setting the **off** (BrOffset) instruction bit field to the corresponding BM bank (For example,

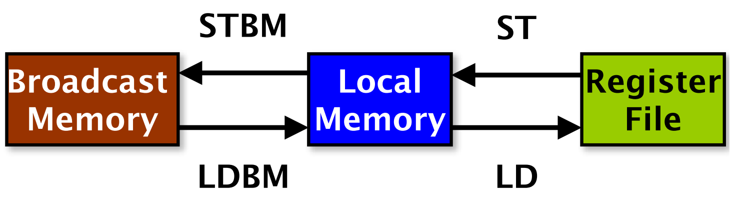


Figure 4.3: Example source and destination of broadcast and local memory transfers.

0b0000 for bank 0, 0b0001 for bank 1 and so on). The **mask** (Mask_load) instruction bit field allows masked data transfers to a specific range of PEs inside the broadcast cluster. That is only a subset of PEs will store the incoming data from BM into their LMs, by comparing their PE IDs to the value specified by **mask**. The starting PE of this masked range is instructed by the lower four bits of **mask**, whereas the number of PEs to be targeted, counting from the starting PE, is dictated by the upper four bits.

Table 4.3 summarizes the different possible combinations of the **BM-Type STBM** opcode. Besides, by using the pseudo-instruction extension mnemonics in this table, it is possible to define two different operations for each base **BM-Type STBM** instruction, in the form of (without the parenthesis):

STBM.(**mode**-extension-mnemonic)

For example, an **STBM.SLM** instruction will copy data from LM into BM, while an **STBM.SRF** instruction will copy data from the register file into BM.

Nevertheless, because they are destined to the Memory Slot of the VLIW PE, **BM-Type** instructions should be packed only into the second VLIW instruction packet slot.

mode (md)	**behavior**	**pseudo-instruction extension mnemonic**
0b11	Copy data from LM to BM	SLM
0b0x or 0bx0	Copy data from register file to BM	SRF

Table 4.3: The **mode** instruction field behavior in a STBM instruction and mnemonics for pseudo-instructions.

Fig. 4.3 depicts most of the possible directions of data transfers using LM-type and BM-type instructions. In summary, a **LD** instruction loads data from LM to RegisterFile, a **ST** instruction stores the data from the second output of the RegisterFile to LM, a **LDBM** instruction loads a chunk of data from broadcast memory banks to LMs and finally a **STBM** instruction stores the data from LM or RegisterFile into the corresponding BM memory bank.

4.3.6 N-Type instructions

63 58	57 50	49 48	47 36	35 20	19 12	11 4	3 0
opcode	un	md|un	lmaddr|un	un	src2|un	ndst|un	nsrc|un

The **N-Type** instructions regroup four different PE neighbor-communication instructions, namely, **NST**, **NPASS**, **NSG**, and **BFLUSH**.

The **NST** (Neighbor STore) is an **N-Type** instruction that takes the data output of a specific communication buffer, selected through the **nsrc** bit field, and stores it into the local memory address marked by the **lmaddr** bit field. The DRAGON architecture supports up to four different PE direct neighbors and four different remote PE neighbors, however, it is up to the designer to select the interconnect topology and number of both direct and remote connected PEs, knowing that DRAGON supports up to eight connected neighbors for each PE. The detailed values for **nsrc**, their corresponding meaning and their pseudo-instruction extension mnemonics are given in Table 4.5. In summary, the **NST** instruction uses only the **opcode**, **lmaddr** and **nsrc** bit fields.

The **NPASS** (Neighbor PASS) is an **N-Type** instruction that allows passing data from one PE to its neighbor. It adds the **ndst** to the bit fields used by **NST**. In fact, it allows to read the output of a neighbor communication buffer, selected through the **nsrc** bit field, in accordance with the Table 4.5. Then, the read value will be stored into the corresponding communication buffer of the neighboring PE, that is selected based on the **ndst** (neighbor destination) bit field.

The **BFLUSH** (Buffer Flush) is another **N-Type** instruction, that is very useful in the context of FPGA implementation. It contains only the **opcode** and the **ndst** bit fields. The latter is used for flushing the corresponding communication buffer by resetting its write and read pointers to zeros.

The last **N-Type** instruction is **NSG** (Neighbor Scatter Gather). This instruction allows scattering data towards neighboring PEs, while in the same time, gathering incoming data from the neighboring PEs into the corresponding communication buffer. The **md** (mode) bit field dictates the source for the data to be scattered as shown in Table 4.4. In addition, by using the pseudo-instruction extension mnemonics in this table, it is possible to define four different operations for each base **N-Type NSG** instruction, in the form of (without the parenthesis): **NSG**.(**mode**-extension-mnemonic)

mode (md)	**behavior**	**pseudo-instruction extension mnemonic**
0b00	Scatter data from ALU output	SA
0b01	Scatter data from FPU output	SF
0b10	Scatter data from LM output	SLM
0b11	Scatter data from RF output	SRF

Table 4.4: The **NSG** instruction **md** bit field behavior and mnemonics for pseudo-instructions.

Besides, because they are destined to the Memory Slot of the VLIW PE, all of the **N-Type** instructions should be packed only into the second VLIW instruction packet slot.

nsrc	**data to be stored or passed**		**pseudo-instruction**
	source	**location**	**extension mnemonic**
0b0000	North Neighbor Buffer	N-FIFO output	NNB
0b0001	West Neighbor Buffer	W-FIFO output	WNB
0b0010	East Neighbor Buffer	E-FIFO output	ENB
0b0011	South Neighbor Buffer	S-FIFO output	SNB
0b0100	Remote North Neighbor Buffer	RN-FIFO output	RNNB
0b0101	Remote West Neighbor Buffer	RW-FIFO output	RWNB
0b0110	Remote East Neighbor Buffer	RE-FIFO output	RENB
0b0111	Remote South Neighbor Buffer	RS-FIFO output	RSNB

Table 4.5: The source communication buffer with each **nsrc** instruction field entry.

Ultimately, the eight bits of **ndst** instruction field, in the descending range [7:0], correspond to the Remote South, Remote East, Remote West, Remote North, South, East, West, North, PE neighbors, respectively. Setting any bit of the **ndst** bit field will scatter the previously read data into the direction of the PE neighbor that corresponds to the position of that bit. The sender PE will write the data coming from the opposite direction of scatter operation into its corresponding communication buffer. That is when scattering for example to North PE, it will write incoming data from its South neighbor into its South buffer. The **ndst** field has the same behavior also in **R-type** instructions.

4.3.7 Virtual relative placement of communication buffers

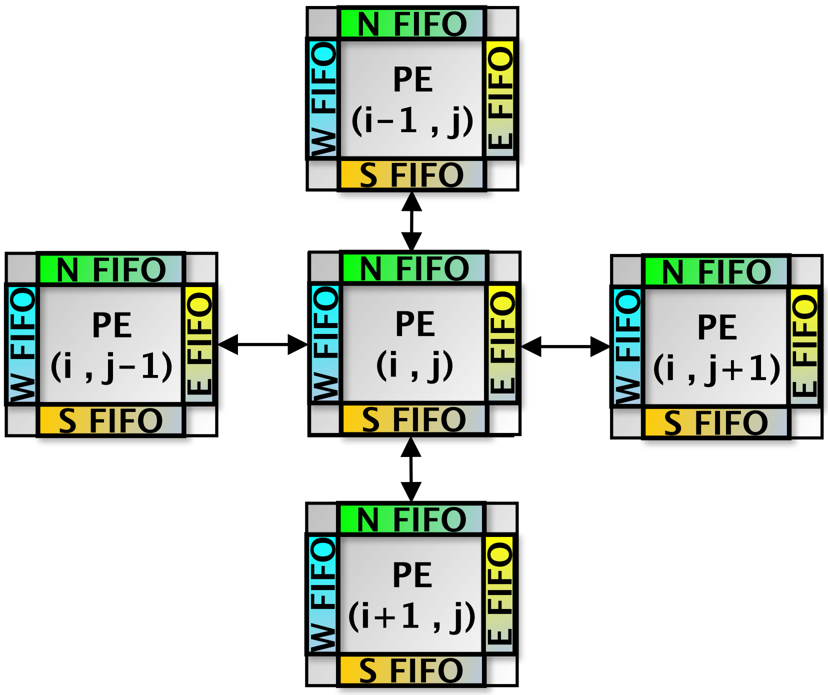


Figure 4.4: Example of virtual relative locations of communication buffers and data exchange directions for scatter/gather operations, with a 2D Mesh interconnect.

An example showing the relative placement of communication buffers and how they are connected to adjacent PEs in a 2D interconnect, is depicted by Fig. 4.4.

The PEs scatter and gather data to/from their neighbors using specific instructions such as N-type instructions (for example NSG instructions which literally means Neighbor-Scatter-Gather). The R-type instructions (register-based compute instructions) can also perform a combined scattering and gathering of data while performing a computation using the ALU or FPU.

A PE can scatter and gather data to/from any of its neighbors as depicted by Fig. 4.4. A simple example of data flow between PEs connected through a 2D interconnect, can be described as follows:

When $PE_{i,j}$ scatters data to the East direction, $PE_{i,j+1}$ store it in its 'W FIFO' communication buffer. Subsequently, $PE_{i,j}$ would gather data from $PE_{i,j-1}$ in its own 'W FIFO' communication buffer. Similarly, when $PE_{i,j}$ scatters its data to the North direction, this data will be stored into the 'S FIFO' communication buffer of $PE_{i-1,j}$. At the same time, $PE_{i,j}$ will store the gathered input data from its South neighbor $PE_{i+1,j}$ in its own 'S FIFO' communication buffer . Ultimately, higher interconnect degrees establish extra remote connections with each PE to support additional directions for exchanged data, while adopting the same concepts of the flow of data between PEs in a 2D interconnect.

4.4 A summary of Instructions Opcodes

Table 4.6: The DRAGON base Instruction Set categories, opcodes and their behavior [3].

Mnemonic	Opcode	Behavior description
NOP	0b000000	No operation
LDimm	0b001001	Load 64-bit value in the RegisterFile
Data processing (64-bit Integer), R-Type		
ADD	0b000001	Integer addition
SUB	0b000010	Integer subtraction
AND	0b000011	Bitwise logical AND
OR	0b000100	Bitwise logical OR
XOR	0b000101	Bitwise logical XOR
SLL	0b000110	Shift Logical Left
SRL	0b000111	Shift Logical Right
MUL	0b001000	Multiply lower 32 bits of both operands
Data processing (Double-precision Floating-Point), R-Type		
FADD	0b010000	Floating-point addition
FSUB	0b010001	Floating-point subtraction
FMUL	0b010010	Floating-point multiplication
FMACCA	0b010000	Floating-point Multiply-Add-Accumulate
FMACCS	0b010101	Floating-point Multiply-Subtract-Accumulate
BM memory transfer operations, BM-Type		
LDBM	0b100010	Load Data from BM to LM
STBM	0b100011	Store data from PE to BM
Register-LM memory transfers, LM-Type		
LD	b100000	Load from LM into Register File
ST	b100001	Store from Register File to LM
Neighbor communication operations, N-Type		
NSG	0b110000	Scatter/Gather to/from adjacent PEs
NST	0b110001	Store from an input buffer into LM
NPASS	0b110010	PASS data from an input buffer to adjacent PE
BFLUSH	0b110011	Reset read/write pointers of fifo input buffers
Controller-scope-limited instructions, C-Type, opcode=**0b111111**		
	funct	
REPEAT	0b000001	Loop for a number of iterations
BNZ	0b000010	Check loop counter then branch if not zero
RDGMEM	0b000011	Configures DMA to pass data from GM to BM
WRGMEM	0b000100	Configures DMA to pass data from BM to GM
STOP	0b000101	Flags the end of a program

As a general categorization, the DRAGON instructions are organized according to their type which depends mainly on the VLIW slot they are destined to (execute or memory slot). The **R-Type** instructions should be on the first packet of the dual-packet VLIW instruction, while **LM-Type**, **BM-Type** and **N-Type** instructions should be on the second packet of the dual-packet VLIW instruction. In a typical micro-architecture implementation, the PE is split into two slots. The first executes computational instructions (**R-Type**) and the second manages memory operations and data movements (**BM-Type**, **LM-Type** and **N-Type**).

Table 4.6 summarizes the different categories of the DRAGON instruction set, while Fig. 4.5 depicts the different formatting of the several categories of instructions.

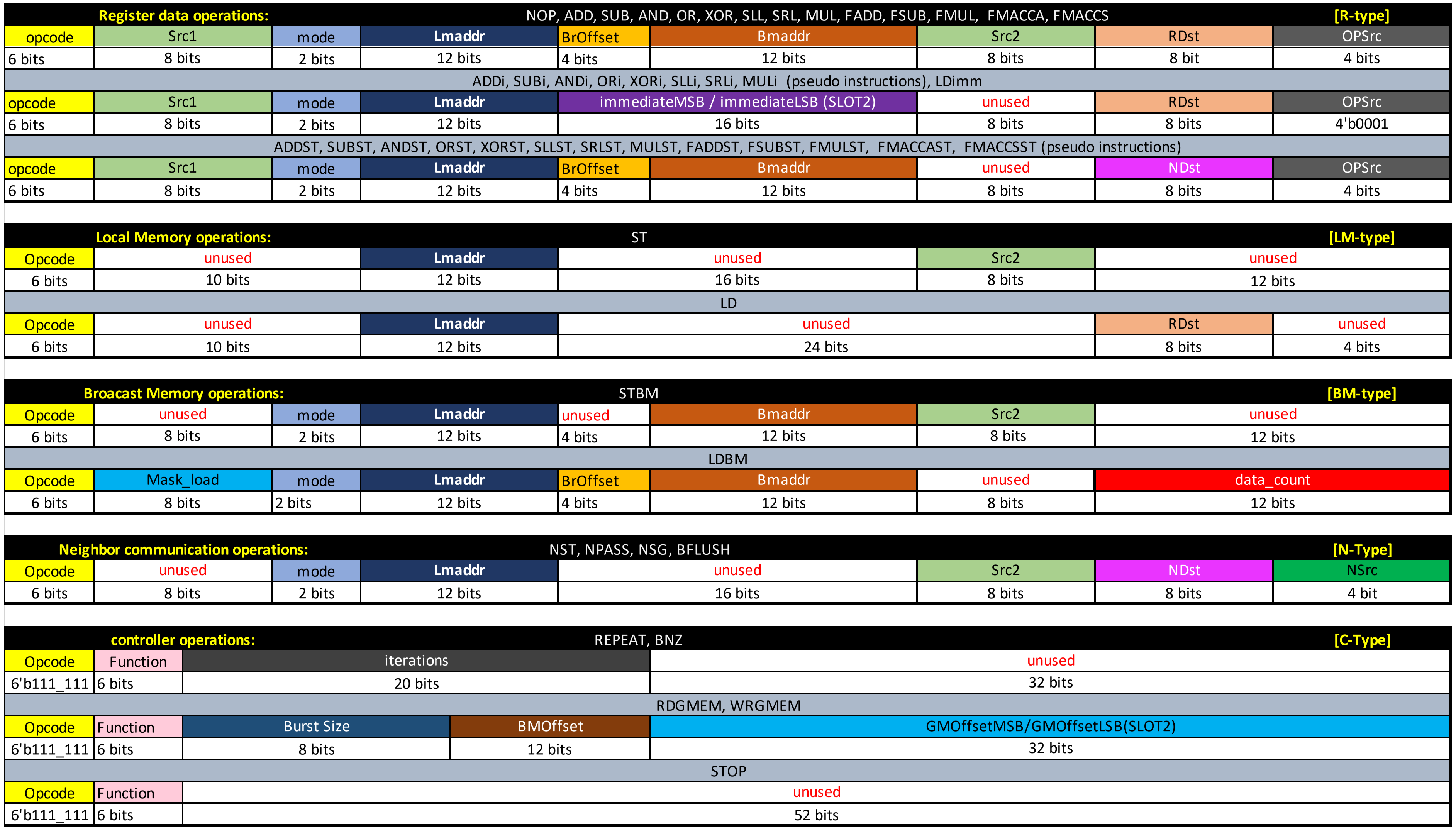
4.5 More on the DRAGON ISA

4.5.1 Pseudo-instructions and Further extensions

DRAGON offers a tremendous amount of configurations for the base instruction set, which considerably increases the number of operations that can be performed. For example, there are only 13 R-Type instructions, however, there are 4 different combinations of the **md** bit field and another 12 different combinations of the second source input operand through the **opsrc** bit field. Consequently, there is a total of 48 different combinations for each single R-Type instruction, when adding the possibilities offered by the **opsrc** and **md** instruction bit fields. This elevates the count of R-Type pseudo-instructions to 624 different possible operations.

Besides, throughout all instruction formats depicted by Fig. 4.5, there are multiple empty bit fields marked as "unused". These bit fields can allow future extensions and specialization of instructions.

Furthermore, the opcode is coded on 6 bits which offers a total of 64 different combinations of instructions, among which, only 24 are currently used (counting the **C-Type** fixed opcode as one of these). This means, there are 40 opcodes that are still available for future extensions of the non-control instructions. On the other hand, the **C-Type** control instructions use just five combination of the **funct** bit field and therefore, there is a total of 59 different combinations available for extending this category of instructions.



Register data operations:			NOP, ADD, SUB, AND, OR, XOR, SLL, SRL, MUL, FADD, FSUB, FMUL, FMACCA, FMACCS					[R-type]
opcode	Src1	mode	Lmaddr	BrOffset	Bmaddr	Src2	RDst	OPSrc
6 bits	8 bits	2 bits	12 bits	4 bits	12 bits	8 bits	8 bit	4 bits
ADDi, SUBi, ANDi, ORi, XORi, SLLi, SRLi, MULi (pseudo instructions), LDimm								
opcode	Src1	mode	Lmaddr	immediateMSB / immediateLSB (SLOT2)		unused	RDst	OPSrc
6 bits	8 bits	2 bits	12 bits	16 bits		8 bits	8 bits	4'b0001
ADDST, SUBST, ANDST, ORST, XORST, SLLST, SRLST, MULST, FADDST, FSUBST, FMULST, FMACCAST, FMACCSST (pseudo instructions)								
opcode	Src1	mode	Lmaddr	BrOffset	Bmaddr	unused	NDst	OPSrc
6 bits	8 bits	2 bits	12 bits	4 bits	12 bits	8 bits	8 bits	4 bits

Local Memory operations:			ST				[LM-type]
Opcode	unused		Lmaddr	unused	Src2	unused	
6 bits	10 bits		12 bits	16 bits	8 bits	12 bits	
LD							
Opcode	unused		Lmaddr	unused		RDst	unused
6 bits	10 bits		12 bits	24 bits		8 bits	4 bits

Broacast Memory operations:			STBM					[BM-type]
Opcode	unused	mode	Lmaddr	unused	Bmaddr	Src2	unused	
6 bits	8 bits	2 bits	12 bits	4 bits	12 bits	8 bits	12 bits	
LDBM								
Opcode	Mask_load	mode	Lmaddr	BrOffset	Bmaddr	unused	data_count	
6 bits	8 bits	2 bits	12 bits	4 bits	12 bits	8 bits	12 bits	

Neighbor communication operations:			NST, NPASS, NSG, BFLUSH				[N-Type]
Opcode	unused	mode	Lmaddr	unused	Src2	NDst	NSrc
6 bits	8 bits	2 bits	12 bits	16 bits	8 bits	8 bits	4 bit

controller operations:				REPEAT, BNZ	[C-Type]
Opcode	Function	iterations		unused	
6'b111_111	6 bits	20 bits		32 bits	
RDGMEM, WRGMEM					
Opcode	Function	Burst Size	BMOffset	GMOffsetMSB/GMOffsetLSB(SLOT2)	
6'b111_111	6 bits	8 bits	12 bits	32 bits	
STOP					
Opcode	Function	unused			
6'b111_111	6 bits	52 bits			

Figure 4.5: The DRAGON Instruction formats [3].

Moreover, when the broadcasting is enabled on **R-Type** instructions, the **src2** bit field is unused and therefore the lower bits value 0b0000 is fixed for the current broadcast scheme, whereas 15 other combinations can be used for different broadcasting schemes that can be proposed in different extensions that may address specific kinds of computation.

Furthermore, there are only 12 used combinations of the **opsrc** bit field in **R-Type** instructions; the remaining possible four combinations are reserved for further extensions of the base DRAGON ISA.

To allow consistent behavior with further ISA extensions, all the unused bit fields in each instruction must be fixed to all-zeros value.

4.5.2 custom-precision computations

The DRAGON ISA intrinsically supports double-precision floating-point and 64-bit long integer compute operations. Nonetheless, the DRAGON ISA targets reconfigurable chips and as such micro-architectural choices and hardware configuration may partially alter the ISA to support custom-precisions with respect to the instruction formatting. In other words, the DRAGON instructions dictates the memory locations and the target operations, however the hardware configuration can set the width of data inside these memories. In fact, through specific hardware implementations of the PE, it is possible to incorporate a packed SIMD behavior inside a compute instruction by performing two single-precision or four half-precision floating-point computations, using a single compute instruction. For example, when the original width of data is set to 64-bit across all memories and input operands, it is possible to pack two 32-bit operands or four 16-bit operands within these data and accordingly split the original MAC FPU and ALU into two or four units, capable of performing the computations that are adequate to the reduced precisions.

4.6 Limitations and primary target application domains

The DRAGON Instruction set provides a minimal set of opcodes that offer sufficient flexibility to address different kinds of computations. The primary target of DRAGON consists of parallel processing problems with an intensive data exchange behavior between neighboring processing elements such as those found in stencil-based calculations. Other applications such as vector or matrix multiplications may be mapped in a way that benefits from the broadcasting feature in each broadcast cluster which minimizes the bandwidth requirement. The multiply-accumulate opcode allows the efficient implementation of operations such as convolutions. Nonetheless, DRAGON remains a SIMD architecture and therefore it can achieve its best performance and efficiency only when targeting applications that can be processed in a parallel manner and maximize the utilization of its multiply-and-accumulate instructions.

Applications that can not be processed in parallel due to irregular memory accesses or conditional divergent executions on PEs require the implementation of a predication scheme or a more sophisticated masking scheme (the baseline ISA supports a limited set of masking in broadcast memory load operations). To provide even more flexibility and target a larger range of applications, DRAGON was designed to accept further instruction set extensions.

4.7 Summary

In this chapter, the DRAGON base ISA is presented and details about each instruction and its related opcode, formatting and expected behavior were described. Besides, some instructions offer some specialization, beyond their opcodes, and therefore, a naming convention has been proposed for pseudo-instructions which may facilitate the programming task.

The proposed ISA aims to achieve the goals of an energy efficient architecture through a multitude of attractive approaches. For instance, it allows the overlap of multiple operations within the same instruction and allows direct data transfers with adjacent processing elements as well as direct operations on data stored into their communication buffers, which may considerably reduce the energy cost related to data movements in general.

Chapter 5

The Programming Model

5.1 Introduction

This chapter provides the necessary details to understand the DRAGON programming model. A system level overview is given to explain the methodology adopted for interfacing the overlay within a heterogeneous computing platform. While this methodology is specific to the target FPGA family, it should provide sufficient information to generalize to other devices. This chapter also shows, through a particular example, the general approach adopted to use the DRAGON assembly opcodes to program the underlying many-core overlay in C or C++ languages and the methodology to build binary executables while relying on mainstream C/C++ compilers. Moreover, examples extracted from an OpenCL-based program are explained to show how the DRAGON overlay can be controlled from an OpenCL-based host. Finally, the use of Verilator in the functional verification process is also described in detail.

5.2 Enhancing the RTL Kernel Model through re-usability

5.2.1 Background

The DRAGON overlay offers a convenient approach for non-hardware experts intending to offload compute tasks from a host system to FPGA accelerators. Basically, overlays can hide FPGA hardware details, by stacking higher abstraction layers that provide designers with simplified views of the underlying physical fabric. Nonetheless, the integration within a heterogeneous platform and the management of the data communication between a host and an FPGA, through an independent or a shared memory space, introduces increased levels of design complexity that keep FPGAs away from mainstream adoption. For example, adopting a PCIe (Peripheral Component Interconnect express) as a means of interconnecting an FPGA with a host, requires extensive expertise and deep knowledge of the underlying communication protocol as well as low-level firmware program details that are usually complicated to debug and maintain.

To overcome these issues, the work in this thesis builds upon an existing framework and transcends it by introducing a novel approach that allows to further abstract the interfacing with a host in an effective and re-usable manner.

5.2.2 A re-usable bitstream of a software programmable overlay

Xilinx Vitis offers a way to control accelerators through an OpenCL function that models a carefully packaged FPGA design. Vitis supports accelerated FPGA kernels that are designed in both HLS (HLS kernel flow) or RTL (RTL kernel flow) and provides the necessary details describing the interface to the kernel flow [112]. In contrast to HLS-based kernels, the use of RTL kernels may require cumbersome low-level FPGA implementation details. Examples of these details include the manual management of memory interfaces and control through AXI protocol, as well as the knowledge of clocks, reset, and specific interrupt signals.

Moreover, RTL kernels are often optimized and implemented for a fixed problem which limits their re-use. In contrast, the work in this thesis aims at transcending the RTL kernel model by implementing an "RTL kernel overlay model" that implements the infrastructure that allows PCIe-based communication between a host and an FPGA, while providing an overlay-based RTL kernel that can be dynamically re-programmed using software instructions.

Fig. 5.1 illustrates the proposed model that harnesses the benefits of the RTL kernel flow model provided by the FPGA vendor while adding a layer of software re-programmability to the kernel that is implemented on the FPGA side. Moreover, this model proposes a methodology to manipulate the host-based OpenCL task offload, by including the transfer of software instructions alongside non-processed data in the offload process.

The RTL kernel flow is used to implement RTL-based FPGA designs. On the host side, this

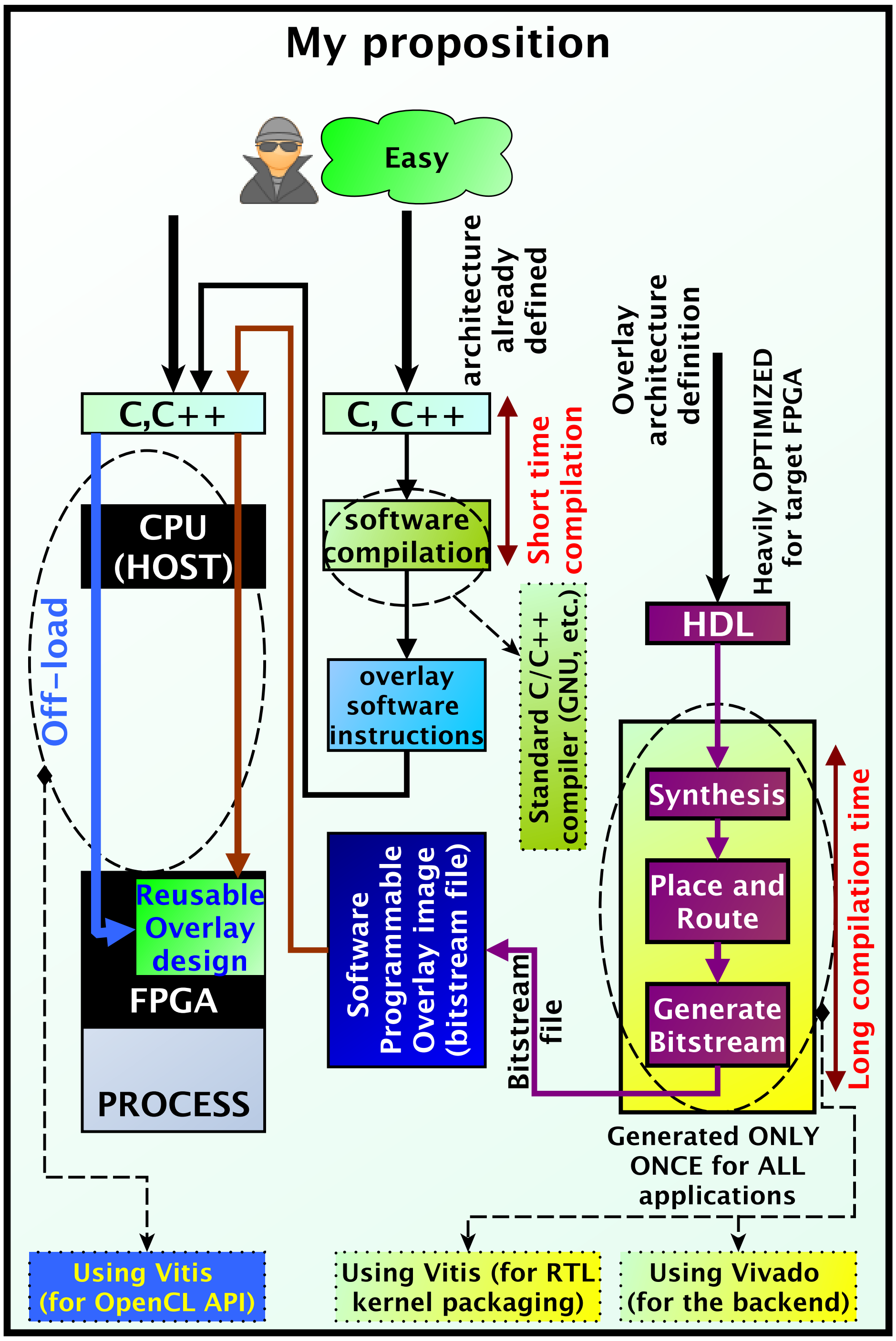

My proposition
Easy
architecture already defined
Overlay architecture definition
C,C++
C, C++
Short time compilation
software compilation
CPU (HOST)
Off−load
Heavily OPTIMIZED for target FPGA
HDL
Standard C/C++ compiler (GNU, etc.)
overlay software instructions
Synthesis
Place and Route
Generate Bitstream
Long compilation time
Reusable Overlay design
FPGA
PROCESS
Software Programmable Overlay image (bitstream file)
Bitstream file
Generated ONLY ONCE for ALL applications
Using Vitis (for OpenCL API)
Using Vitis (for RTL kernel packaging)
Using Vivado (for the backend)

Figure 5.1: Overview of the proposed dynamically re-programmable approach.

design is seen as a software function, that can be called in a host-based program as a standard OpenCL task. As such, the RTL kernel flow imposes packaging the design as an IP (Intellectual Property) with an interface that corresponds to the one expected by Vitis [112]. This FPGA-based interface must contain an external memory interface alongside scalar control signals whose details are explained in later sections.

On the other hand, the software function model that abstracts the RTL kernel design should have pointer and scalar arguments that match the global memory banks (which can be implemented either using HBM2 memory or general on-chip memories such as URAM) start addresses and control signals in the interface of the packaged DRAGON overlay design. This overlay is carefully designed using SystemVerilog language to achieve high quality of results. Xilinx Vitis is used to package the design following the RTL kernel flow, then later synthesize, implement and generate the FPGA bitstream file, which relies on Xilinx Vivado in the background.

Besides, the DRAGON overlay is a custom ISA software programmable overlay. Consequently, a software program is written using C or C++ languages and inlines the DRAGON ISA assembly level opcode functions within. These are C-based functions that abstract each opcode functionality and serve as an API (Application Programming Interface) to program DRAGON. Each call of one of these functions from a C/C++ program will output the equivalent hexadecimal expression of the underlying opcode alongside the following instruction fields. Therefore, the compilation and subsequent separate execution of this program on a host platform will generate the executable file that needs to be transferred alongside non-processed data to the FPGA side. A standard C or C++ compiler can be used for compiling this program prior to generating the executable file whose contents will be loaded in the IM (Instruction Memory) of DRAGON.

On the host side, an OpenCL program is written to connect the host to the target FPGA. OpenCL buffers have to be created in a way that matches the used global memory buffers on the RTL-kernel packaged DRAGON overlay. The OpenCL function that models the overlay is enqueued for execution which will send the non-processed data as well as the overlay program instructions to the connected FPGA, execute these instructions on the DRAGON overlay and then store the processed data results back into the corresponding previously allocated host buffers. Should new non-processed data and new program instructions be loaded into the FPGA, the same bitstream can be re-used and the FPGA need not to be reconfigured. In the background, Xilinx Vitis provides the necessary infrastructure to use the PCIe-based communication between the host and the FPGA.

5.2.3 Details of the overlay integration within the heterogeneous computing platform

The scalar arguments of the software function abstracting the DRAGON overlay design are the parameters that allow the configuration of this overlay, such as a request to reload program instruction from the GM to the IM, or the total size in bytes of these instructions. These scalar arguments can be transferred from the host to the overlay Control Unit through an AXI-Lite Interface. In addition, more scalar arguments can be defined in the software function to implement custom-defined behavior of the overlay, however, they should be accounted for in the implemented Control Unit.

The DRAGON overlay implements an AXI4-compliant master interface that is used for data exchange with the GM. The DRAGON overlay is also packaged as an RTL kernel alongside its XML (Extensible Markup Language)-based interface definition file which provides information about the top level RTL module interface.

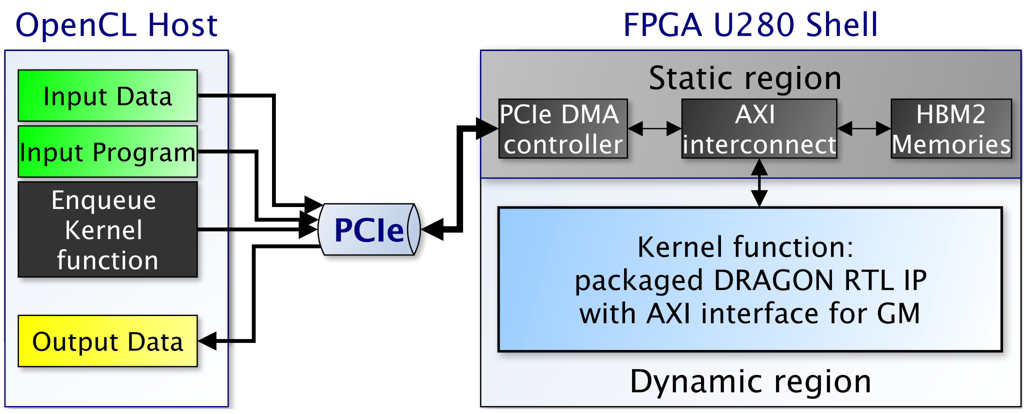


Figure 5.2: Overlay Integration within a heterogeneous computing platform [3].

The Vitis framework automatically creates the FPGA shell (static region) that implements the AXI interconnect between the provided AXI4 design interfaces and the global memory (can be implemented as HBM memory) banks. In addition, the PCIe DMA is automatically created as the required infrastructure to allow data movement between the host-side and the FPGA-side as depicted by Fig. 5.2. While depicted as separate in Fig. 5.2, the input and output data may actually share the same buffers.

The remaining FPGA area is called the dynamic region and is the part of the FPGA that can be used by the designer to freely place the RTL kernel and its different modules. In contrast,

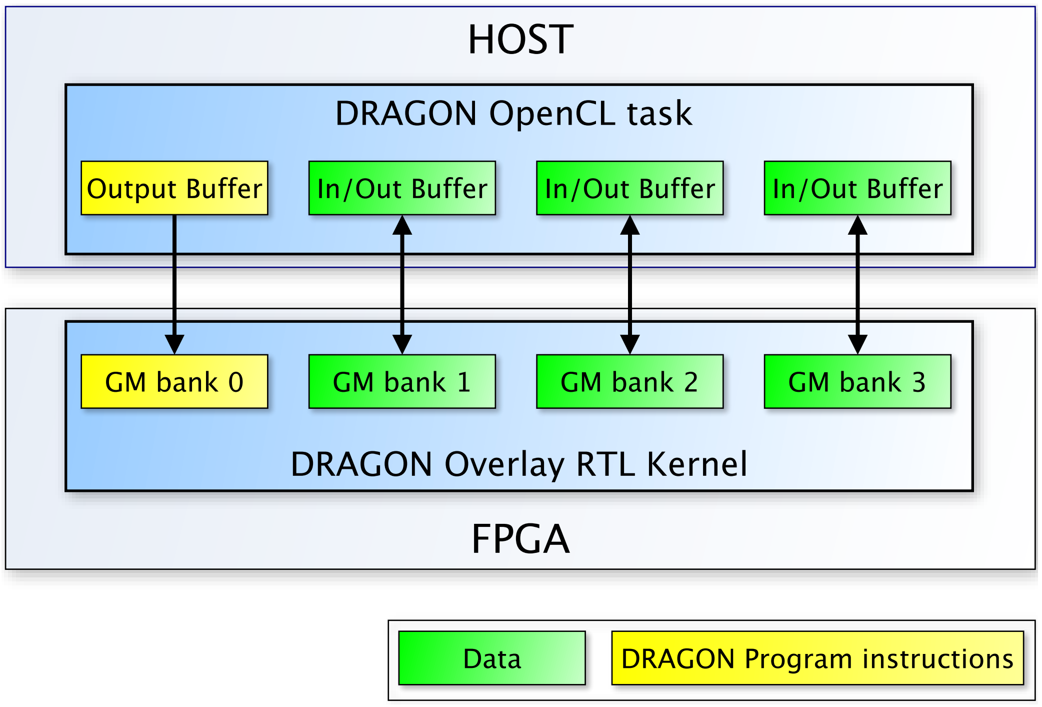


Figure 5.3: DRAGON host-side/FPGA-side memory space mapping.

the static region is a private area, inaccessible directly to the designer.

On the host side the OpenCL creates buffers to hold the data that need to be exchanged with the FPGA side. Standard RTL kernels are not software-programmable and therefore need only to exchange input and output data with the host side. In contrast, DRAGON is software-programmable, therefore, it exploits the RTL kernel model by allocating a buffer in Output-only mode to transfer the instructions that will be executed by the overlay without needing to reload a new bitstream file for every new application.

An example mapping between host buffers and FPGA GM banks is depicted by Fig. 5.3.

The OpenCL program can transfer both instructions and input data from its host-side buffers to the FPGA-side DRAGON overlay packaged as an RTL kernel.

5.2.4 Detailed control and execution flow

Fig. 5.4 shows a detailed example of operation for the implemented control and execution methodology.

The operation of DRAGON can be started by enqueuing the equivalent DRAGON OpenCL software function (OpenCL task) for execution. When this task is granted permission by the

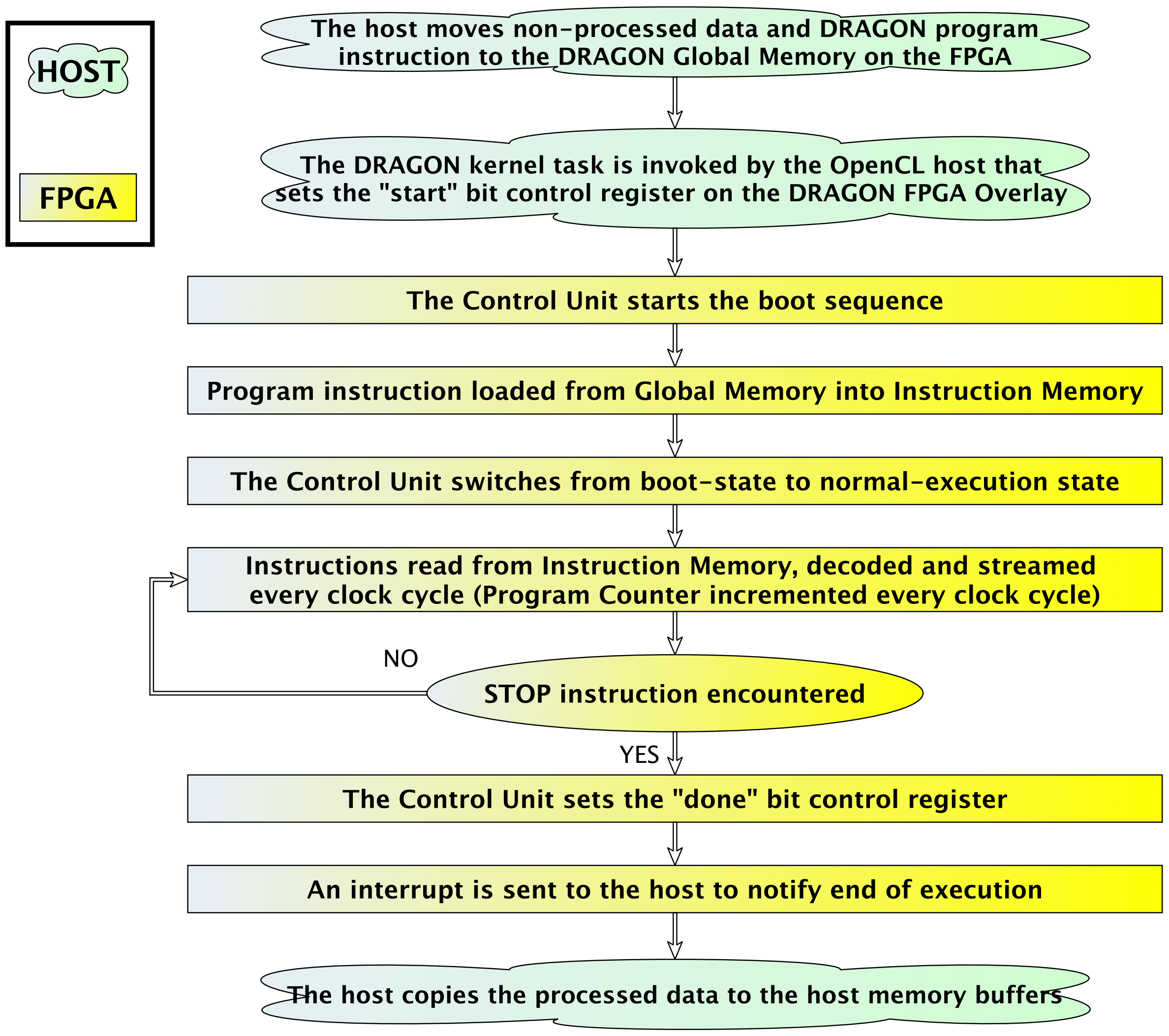

HOST
FPGA
The host moves non-processed data and DRAGON program instruction to the DRAGON Global Memory on the FPGA
The DRAGON kernel task is invoked by the OpenCL host that sets the "start" bit control register on the DRAGON FPGA Overlay
The Control Unit starts the boot sequence
Program instruction loaded from Global Memory into Instruction Memory
The Control Unit switches from boot-state to normal-execution state
Instructions read from Instruction Memory, decoded and streamed every clock cycle (Program Counter incremented every clock cycle)
NO
STOP instruction encountered
YES
The Control Unit sets the "done" bit control register
An interrupt is sent to the host to notify end of execution
The host copies the processed data to the host memory buffers

Figure 5.4: Example of the execution flow on the FPGA-based overlay.

host to start execution, after the data allocated in the host-side buffers are transferred to the GM (into the FPGA), a start signal (a single-bit register) is set into an AXI-Lite control interface (on the FPGA) that is subsequently sent to the DRAGON Control Unit to launch the boot sequence waking the overlay from its idle state.

The boot sequence allows downloading program instructions from the GM into the IM. Nonetheless, in situations where the program is already stored in the IM and has to be reused, this boot sequence can be bypassed, by setting a corresponding scalar control argument in the OpenCL software function that models the overlay interface.

After the boot step has finished, the Control Unit jumps to the normal-execution state of the overlay, where a program pointer is incremented to read instructions from the IM, decode them and then issue the different decoded streams for execution.

Following **RDGMEM** instructions, the data can be moved from the GM through DMAs to their corresponding BMs, processed into the accelerator part and finally, the results of the computation are stored back to GM, using **WRGMEM** instructions.

The end of program execution is signalled when a **STOP** instruction is encountered. At that time, the DRAGON Control Unit notifies the host about data processing completion by setting a specific register in an AXI-Lite control interface. This allows the host to move back the results of the computation from the GM memory banks (which can be implemented using HBM, DRAM or even URAM memories) to its corresponding previously allocated buffers.

5.3 Host-Side Programming

Fig. 5.5 shows the mapping between the host and FPGA buffers using the abstract model of the RTL kernel overlay through an OpenCL prototype function and its pointer and scalar arguments.

The use of FPGAs in heterogeneous computing platforms is gaining momentum as these devices have proved to be a potential accelerator for multiple kinds of workloads. While FPGA vendors provide support for the integration of FPGAs in such platforms, their proposed model provide limited support for programmable accelerators. Mainly, their model consists of an accelerator running on the FPGA-side and a control program running on the host-side. The control program recognizes the external memory interface of the accelerator and transfers the non-processed data, instructs the accelerator to process them and then transfers back the processed data to the host-side memory.

The DRAGON execution model leverages this approach to provide a general methodology to control programmable accelerators, without worrying about low-level implementation details. As such, alongside the memory buffers for non-processed data, the control program on the host-side, allocates an extra OpenCL buffer, on the FPGA, in read-only mode, for the program instructions that need to be executed by the programmable FPGA-based accelerator (here the DRAGON overlay).

For example, line 4 to 6 of Listing 5.1 show how to allocate data buffers that will be used by global memory to transfer data for each broadcast cluster's broadcast memory banks. These buffers can be read or written into by the DRAGON overlay and therefore a **CL_MEM_READ_WRITE**

Listing 5.1: Creating OpenCL memory buffers on the FPGA side

```
// These commands will allocate memory on the FPGA. The cl::Buffer objects can
// be used to reference the memory locations on the device.
// Creating Buffers
for (int it=0;it<nBCs;it++){
  OCL_CHECK(err, buffer_rw[it] = cl::Buffer(context, CL_MEM_READ_WRITE |
      CL_MEM_EXT_PTR_XILINX |CL_MEM_USE_HOST_PTR,sizeof(DATATYPE) * dataSize, &
      inoutBufExt[it], &err));
}

  //program bank
  //------------------------
OCL_CHECK(err, buffer_rw_hex_prog = cl::Buffer(context, CL_MEM_READ_ONLY |
    CL_MEM_EXT_PTR_XILINX |CL_MEM_USE_HOST_PTR,sizeof(uint32_t) * ProgSizeWords, &
    outBufExt_hex_prog, &err));
```

Listing 5.2: Setting the DRAGON kernel arguments

```
// Setting the DRAGON kernel Arguments
// ------------------------------------------------------------
int n=0;
bool prog_reload = RELOAD;
//scalar arguments
OCL_CHECK(err, err = krnls.setArg(n, prog_reload));
OCL_CHECK(err, err = krnls.setArg(++n, ProgSizeBytes));
//pointer arguments
for (int it=0;it<nBCs;it++){
  OCL_CHECK(err, err = krnls.setArg(++n, buffer_rw[it]));
}
OCL_CHECK(err, err = krnls.setArg(++n, buffer_rw_hex_prog));
// ------------------------------------------------------------
```

Listing 5.3: Moving data between FPGA and the host and invoking the DRAGON kernel for execution

```
// Copy input data to DRAGON Global Memory
 // ------------------------------------------------------------
 OCL_CHECK(err, err = q.enqueueMigrateMemObjects(
               {buffer_rw, //data buffers
                buffer_rw_hex_prog}, //instruction buffer
                0 /* 0 means from host*/));
 q.finish();

 // Invoking the DRAGON kernel
 // ------------------------------------------------------------
 OCL_CHECK(err, err = q.enqueueTask(krnls));
 q.finish();
 std::cout<<"passed queue"<<std::endl;

 // Copy Computed Result from DRAGON Global Memory in the FPGA to Host Local Memory
 // -------------------------------------------------------------
 OCL_CHECK(err, err = q.enqueueMigrateMemObjects(
                     {buffer_rw,buffer_rw_hex_prog},
                      CL_MIGRATE_MEM_OBJECT_HOST));
 q.finish();
```

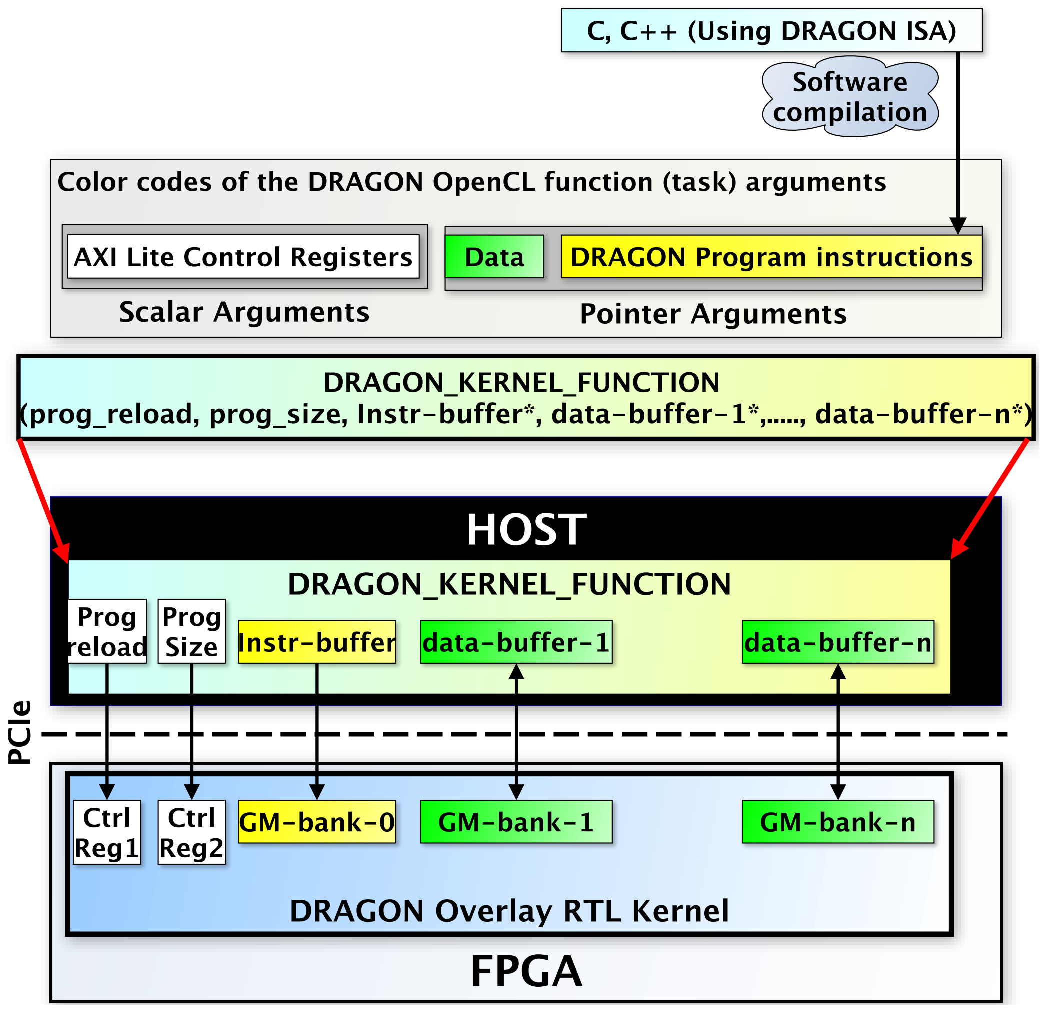

C, C++ (Using DRAGON ISA)
Software compilation
Color codes of the DRAGON OpenCL function (task) arguments
AXI Lite Control Registers
Data
DRAGON Program instructions
Scalar Arguments
Pointer Arguments
DRAGON_KERNEL_FUNCTION
(prog_reload, prog_size, Instr-buffer*, data-buffer-1*,....., data-buffer-n*)
HOST
DRAGON_KERNEL_FUNCTION
Prog reload
Prog Size
Instr-buffer
data-buffer-1
data-buffer-n
PCIe
Ctrl Reg1
Ctrl Reg2
GM-bank-0
GM-bank-1
GM-bank-n
DRAGON Overlay RTL Kernel
FPGA

Figure 5.5: Abstracting the DRAGON overlay as an OpenCL function.

flag was selected. In contrast, the extra buffer allocated for transferring program instructions to the programmable DRAGON overlay sets a read-only flag (**CL_MEM_READ_ONLY** flag) (line 10 of Listing 5.1).

In this model, the DRAGON overlay is considered as an OpenCL (RTL-based) kernel. Therefore, prior to kernel call and execution, a number of arguments must be set by the host. Examples of these arguments are given by Listing 5.2 and indicate whether DRAGON needs to reboot (reload program instructions from GM) (line 6 of Listing 5.2), provide the total size of the program in Bytes (line 7 of Listing 5.2), and provide the list of base pointers for each FPGA-based GM memory bank for data (line 9 to 11 of Listing 5.2) and instructions (line 12 of Listing 5.2).

Then, the OpenCL host code copies the non-processed data and instructions to the GM memory on the FPGA-side (line 3 to 7 of Listing 5.3), invokes the DRAGON kernel for execution (line 11 and 12 of Listing 5.3), and finally, copies back the processed data from the FPGA-side to the host-side memory (line 17 to 20 of Listing 5.3).

5.4 FPGA-side Programming

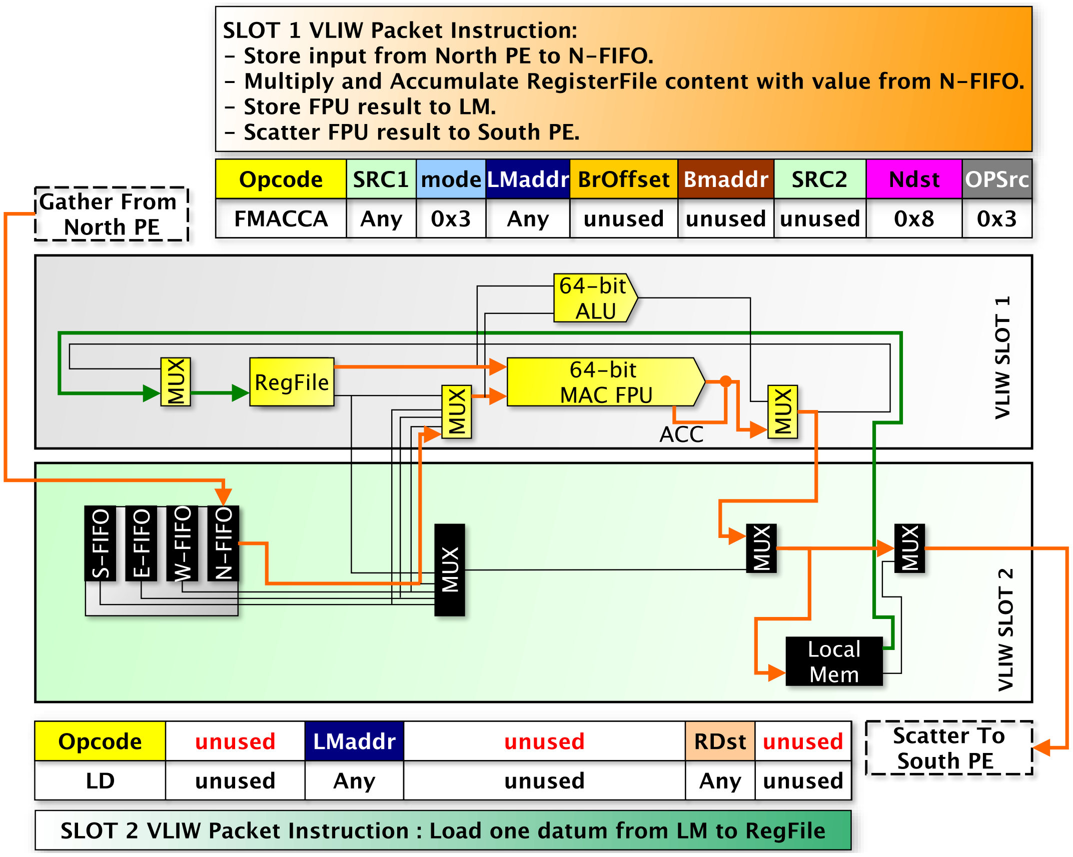


Figure 5.6: VLIW in action: more operations with less instructions.

The DRAGON ISA extends the RISC (Reduced Instruction Set Computer) register-register approach by proposing a more inclusive model that allows operations on multiple operand sources. For example, this model offers the possibility of operation on operands that are incoming from the register file outputs, but also other sources, including neighboring PEs (through communication buffers), or the broadcast memories of the local PE cluster.

The memory locations on each instruction, use direct addressing mode and together with register locations they are transferred after instruction decoding to each PE in the overlay, through dedicated instruction streams. The DRAGON overlay executes SIMD instructions and therefore it targets parallel processing applications. The adopted SIMD model dictates that the same instructions are broadcasted to every PE that execute the underlying operations on either the locally stored data or even the cluster shared data stored in the broadcast memories. This implies that the program is agnostic to the number of BCs in the overlay (consequently the amount of PEs as well); thus, increasing the overlay size (amount of BCs) would not impact the size of the executable code, provided the problem size scales proportionally.

Moreover, The VLIW design approach boosts parallel processing capabilities by combining multiple packets of instructions into a single larger instruction. Furthermore, these packets may in turn, combine multiple operations that are executed in the same clock cycle. A basic VLIW micro-architecture model was proposed in [3] and [4] and it proposes two packets, one for memory operations and the other for compute instructions. The goal is the efficient overlapping of memory operations and data transfers with effective computations. Using this approach allows the sustained performance to reach levels near its theoretical peak and significantly reduces the overhead time of data movement which improves the energy-efficiency. For example, Fig. 5.6 illustrates a scenario of this overlap where the cycle-time cost related to the data transfers between PEs and the load-store data movements, can be hidden by combining these operation with other effective computations. As a result, the PE can effectively perform six operations (Multiply and accumulate is counted as two operations) using a single large VLIW instruction that contains two packet instructions.

5.5 Machine Code Executable Generation

Developing a compiler that targets a new ISA is a complex task, in particular, if the target micro-architecture adopts a VLIW programming model. Instead, it is possible to rely on existing C compilers such as GNU GCC to generate machine code for DRAGON, by embedding its instruction opcodes into high-level C-functions whose arguments are the bit fields of each instruction. Consequently, a typical DRAGON program is written in C language and uses its high-level constructs while in-lining DRAGON ISA instructions within.

To allow this possibility, all C-based function prototypes describing the low-level assembly instructions have been written and collected in a single header file. These prototypes shift the input arguments into their specific position into the instruction. An extra argument has been added to all these function to write the translated function hexadecimal sequence outcome into the DRAGON executable output file. An example prototype function of the ADD opcode is

Listing 5.4: An example prototype function of the ADD opcode

```

//-----Add content of regs RSRC1 and RSRC2 and store sum in RDST----------------------
inline void ADD(unsigned long RSRC1, unsigned long MODE, unsigned long LMADDR, unsigned
     long BROFFSET, unsigned long BMADDR,unsigned long RSRC2,unsigned long RDST,unsigned
     long OPSRC, ofstream& f){
   unsigned long ADDopcode = opADD ;
   INSTRUCTION = (ADDopcode<<58) | (RSRC1<<50) | (MODE<<48) | (LMADDR<<36)|(BROFFSET
       <<32) | (BMADDR<<20) | (RSRC2<<12) | (RDST<<4) | OPSRC;
   INSTR_MSB = INSTRUCTION >> 32 ;
   INSTR_LSB = INSTRUCTION & (0x00000000FFFFFFFF) ;
   f << std::hex << std::setw(8) << std::setfill('0')<< INSTR_LSB <<endl;
   f << std::hex << std::setw(8) << std::setfill('0')<< INSTR_MSB <<endl;
}
```

shown in Listing 5.4. In this listing, the generated instruction is split into two 32-bit MSB and LSB parts. This is to facilitate the functional verification using Verilator, which deals with large bit fields as chunks of 32 bits.

Later, every inline call of a given opcode into the main C program, is expected to write the translated binary sequence to the same file. After the C program is compiled using a standard C compiler, it generates an executable program that, once executed, will provide the final file, containing hexadecimal instructions that can be executed by the DRAGON overlay. The hexadecimal instructions contained in this file may be later transferred to the FPGA overlay through the dedicated OpenCL buffer.

An example showing the use of these C-based functions that abstract the assembly opcode, inside a C program, can be seen in line 2,4,6 and 8 of Listing 10.1. The compilation of the C program including these functions and its subsequent execution would write the hexadecimal instructions resulting from each call to a C-based opcode function, into a separate file (f). This file represents the executable file containing all the instructions that will be sent to the DRAGON overlay through the dedicated OpenCL buffer. These instructions will be later loaded from the corresponding FPGA GM buffer (can be implemented in an HBM bank) to the IM where they can be later read, and decoded by the Control Unit of the DRAGON overlay, for subsequent operations execution.

5.6 Functional Verification Using Verilator

FPGA debugging capabilities are limited by the set of options available from each vendor. In the context of many-core processors, tracking and debugging functional issues may quickly become infeasible with direct emulation on target FPGA due to the lack of infrastructure or extra resources to store intermediate execution results. Verilator is a powerful tool that is often used by chip companies for modeling and early verification purposes. This tool translates a fully synthesizable Verilog code (SystemVerilog is mostly supported as well) into a C++ clock-cycle-based model, that can provide levels of magnitude in simulation speed against event-based HDL simulators. Furthermore, it facilitates the debug process by offering the entire arsenal of C++ high-level constructs to write advanced testing scenarios. Verilator was heavily used during the process of HDL-based design of DRAGON, to keep track of functional issues. These issues may mostly arise during the AXI-based data transfer (read and write operations) between the BM and the GM banks. To avoid these issues, two C-based prototype functions were created to emulate the behavior of AXI slave memories (GM banks which can be implemented in HBM banks) and connected to the top module interface of DRAGON.

As such, it became possible to execute machine code on DRAGON using Verilator, before compiling the design sources to an FPGA bitstream and executing it directly on the target hardware.

5.7 Summary

This chapter summarizes the programming model concepts from different perspectives, including host-side control, FPGA-side programming, executable code generation as well as the functional verification of the proposed overlay using high-level CPP-based test-benches. In summary, the host controls the operation of the DRAGON overlay through an OpenCL-based program. The host manages the allocation of memory buffers and then provides the necessary means to transfer non-processed data along with the executable instructions to the FPGA-based DRAGON overlay which executes these instructions, processes these data and later notifies the host about the processing completion and transfers back the processed data from GM banks to the corresponding memory buffers in the host-side. The generation of executable instructions of DRAGON consists of creating a C-based program, where the DRAGON ISA opcodes are inlined through the use of C-based opcode function prototypes. Compiling and executing this program will cause the C-based opcode functions to write the contents of each instruction in hexadecimal format to a special file that will be used as an executable input to the DRAGON overlay in subsequent steps.

Part III

Hardware Part: FPGA-based accelerator virtualization

Chapter 6

The DRAGON Many-Core-Processor Overlay Architecture: A General Overview

6.1 Introduction

This chapter introduces the general architecture overview of the proposed DRAGON many-Core overlay as well as its different hardware building blocks. This chapter also serves as a basis for the different micro-architectures, which will be discussed in the chapters that follow.

6.2 The DRAGON System-Level Architecture

DRAGON (Dynamically Reprogrammable Architecture for Gather-scatter Overlay Nodes) is a software-programmable many-core-processor overlay architecture targeting reconfigurable chips and aiming for both high computational performance and high energy-efficiency. A general overview of DRAGON is depicted by Fig. 6.1. The architecture itself is split between two major parts operating in tandem, namely, the Controller and the Accelerator.

The Controller orchestrates the execution by controlling the array of PEs (Processing Elements) implemented in the Accelerator. The Controller's main modules are the Sequencer depicted by Fig. 6.2 and the DMA (Direct Memory Access) engine whose detailed behavior is explained in subsequent sections. First, the Controller loads program instructions from the GM (Global

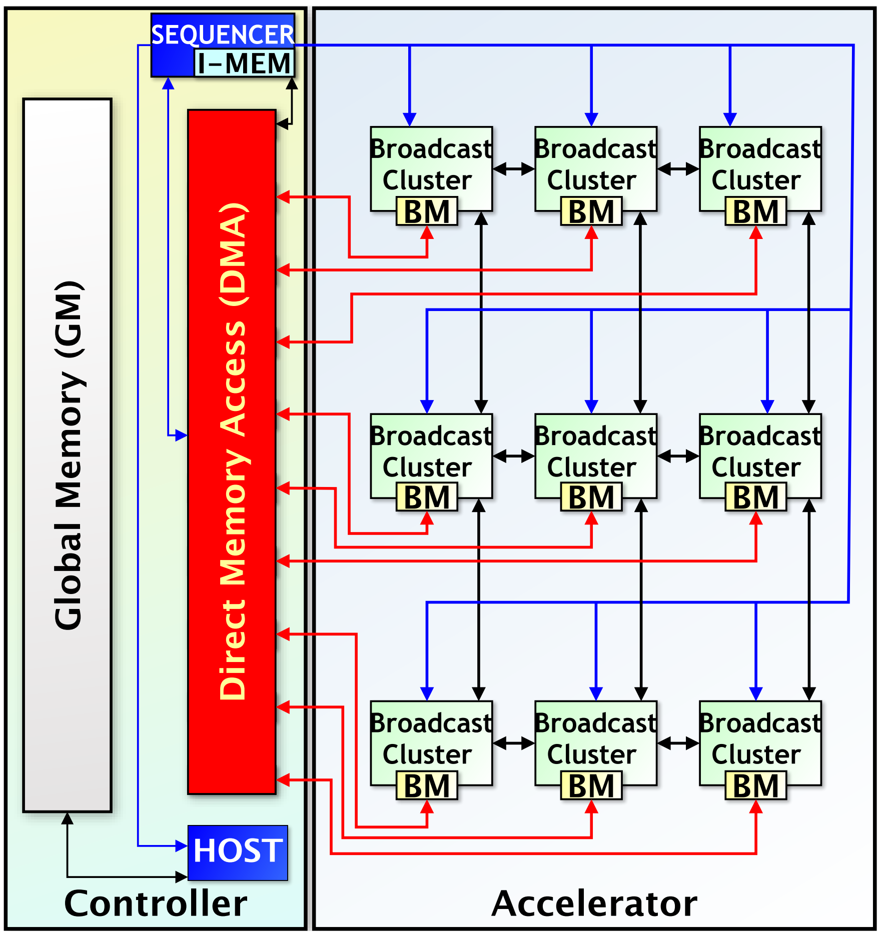


Figure 6.1: The DRAGON system level general architecture.

Memory) into the IM (Instruction Memory). Then, it decodes these instructions and issues multiple dedicated instruction streams towards the Accelerator and the DMA. The Controller also moves the data back and forth between the Accelerator and the GM through one or more DMA engines. The exact operation on these DMAs is specified by a particular type of instructions called **C-Type**, that is discussed in Chapter 4. The Controller also allows optional interfacing with a host system.

On the other hand, the Accelerator is an array of PEs, that are clustered in groups called BCs (Broadcast Clusters). A BC embeds a 4x4 array of PEs, that are connected through some interconnect topology of some dimension, that is extended further by connecting PEs across the BCs as well. Nonetheless, almost any topology and dimension of interconnect can be adopted unless the required number of connected neighbors to a certain PE exceeds eight, which is a limitation set by the current base DRAGON ISA.

The BCs provide a memory interface called BM (Broadcast Memory) that acts as a buffer that interfaces the GM and the array of PEs inside each BC cluster. This buffer decouples data accesses to GM from the execution operation on the PEs.

6.3 Parallel Processing Models

6.3.1 The SIMD Execution Model

The SIMD (Single Instruction Multiple Data) parallel processing paradigm is adopted by the DRAGON overlay architecture. This approach is one of the four classes of computing as defined by Flynn's taxonomy [107, 113].

In the SIMD paradigm a single instruction is broadcasted towards multiple Processing Elements that will concurrently execute it, using different sets of data that are stored locally [12].

Many reasons motivated the adoption of this paradigm. First of all, the fine-grained nature of FPGA is a great fit for such paradigm. In fact, FPGAs offer several kind of hardware resources that are duplicated in hundreds or thousands of modules (such as memories, LUTs and DSPs). These modules are spread across its physical fabric, mostly in a similar structure. These modules can effectively be combined and used to execute similar operations on different data.

Moreover, the programming model is attractive due to its efficiency, and ease of use. In fact, the hardware parallelism is exposed to the programmer and allows multiple applications to benefit from a straightforward mapping of software instructions to hardware compute units.

Finally, and most importantly, the SIMD paradigm simplifies hardware design implementations by requiring a substantially reduced amount of control signals and logic and consequently it allows these designs to deploy more computing blocks and efficiently utilize the available FPGA resources, thus, effectively boosting the overall energy-efficiency. In fact, the removal of redundant control logic is the main reason for resource utilization and power consumption reduction.

From a software perspective, most of the large-scale computational workloads, adopt this paradigm for the improved throughput and the ease of programming.

6.3.2 The VLIW Instruction Model

The VLIW (Very Large Instruction Word) model consists of issuing multiple instructions in packets combined in turn into a single larger one, that is capable of performing several concurrent operations in the same clock cycle.

This paradigm is rather considered a micro-architectural design approach rather than a parallel processing architectural style[11]. The instruction parallelism in VLIW-based processors is explicitly exposed to the software designer at the architecture level. In fact, instead of relying on the compiler to re-organize the assembly-level operations to extract the instruction-level parallelism, the VLIW design philosophy relies on the programmer to efficiently schedule and issue tasks for execution on the VLIW processor. This certainly makes programming more challenging and requires deep understanding of the hardware architecture, however it allows extremely efficient utilization of the underlying hardware parallel processing capabilities

Here, The VLIW style of DRAGON consists of partitioning the instruction into two packets that are destined to be executed concurrently by two different slots in the PE.

The first slot in a DRAGON PE performs purely computational instructions, while the second slot performs data management between PE neighbors and local memory operations. Consequently, this adopted VLIW style, improves the energy-efficiency by providing performance levels close to the theoretical peak, through efficient overlapping of computations and data movements.

6.3.3 A Software-coupled Hardware-decoupled Access Execute Approach

In many-core processing systems, the performance bottleneck is not the computation itself, but rather the movements of data between their memory locations and their respective execution units. The DRAGON programming model aims to solve this bottleneck through efficient separation of the execution logic from the data movement management logic. To realize this goal, the DAE (Decoupled Access Execute) paradigm is adopted.

This approach was proposed in [13] and suggested adopting a pair of split streams of instructions that use hardware-based queues for interacting with each other. At that time, this approach provided a substantial decoupling of data movements and execution. Nonetheless, for modern-day complexity level of many-core processors, it may be quite challenging for both hardware and software designers alike to implement such an approach. In fact, the decoupling of execution and data movement through two different kind of instruction streams would require relying on one or more separate compilers to generate the different executable binary code for each stream.

Worse yet, the same paper states that in the probable unfortunate event where a deadlock occurs, the program requires a purge and a deadlock error is raised and flagged.

In contrast, while DRAGON implements a similar model, it proposes an improved logic that avoids these issues while keeping the benefits of decoupling data access from execution.

In fact, DRAGON combines both control streams into the same program while effectively decoupling the hardware part responsible for the execution, from the one responsible for data transfers to/from global memory. Thus, the suggested name of a Software-coupled Hardware-decoupled Access Execute Approach. In this approach, the same sequencer issues both sides instruction streams, where one stream controls the execution process on the accelerator side, while the other stream configures the DMA engines to allow the accelerator to access the input data.

Consequently, this proposed model guarantees deadlock-free synchronization between both sides instructions streams, while the executable binary can be generated through a single program compiler. Better yet, this removes the need for hardware queues by using the same unified program to control the intermediate data buffer through the use of dual-sided BMs (Broadcast Memories) that can be randomly accessed by both the Controller or the Accelerator.

6.4 The Controller

6.4.1 The Sequencer

The sequencer is the brain of the Controller part and regroups the CU (Control Unit) and the IM (Instruction Memory). The program instructions to be executed by DRAGON are located in the IM and are initially loaded from GM (Global Memory) through a DMA (Direct Memory Access) engine, that is configured and controlled by the CU. Here, the detailed operation of the sequencer will be explained by revealing the behavior of all of its components.

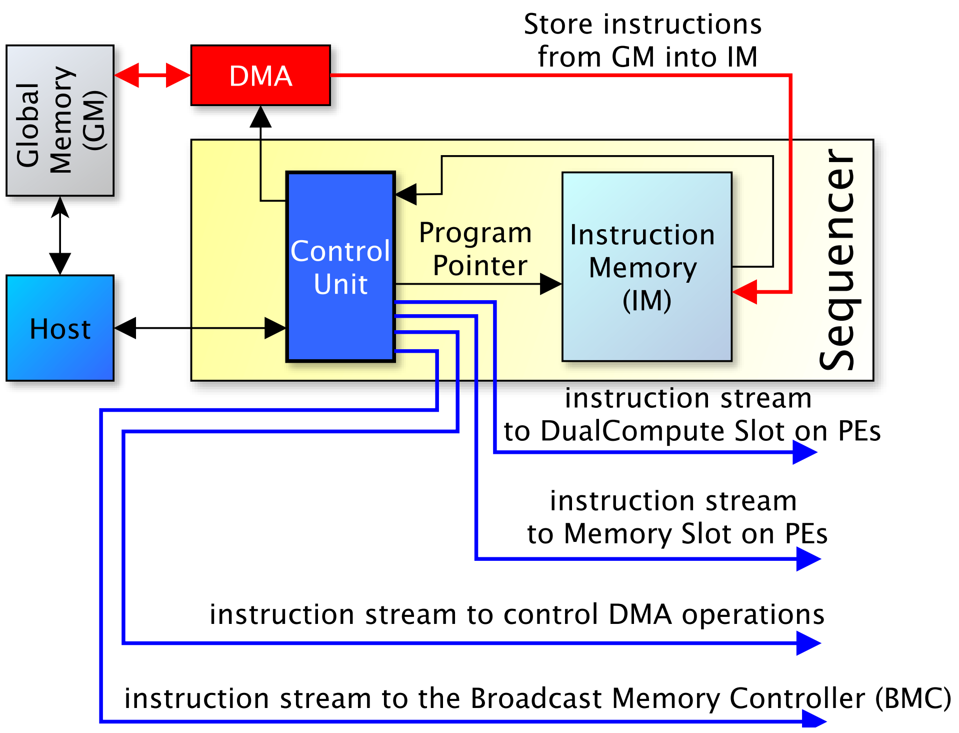

Store instructions from GM into IM
Global Memory (GM)
DMA
Control Unit
Program Pointer
Instruction Memory (IM)
Sequencer
Host
instruction stream to DualCompute Slot on PEs
instruction stream to Memory Slot on PEs
instruction stream to control DMA operations
instruction stream to the Broadcast Memory Controller (BMC)

Figure 6.2: The DRAGON sequencer general architecture.

6.4.1.1 The Instruction Memory

The IM (Instruction Memory) holds the program instructions that are loaded from the GM. Micro-architecture hardware implementations are free to set the size and the number of ports of the IM, however, they must ensure that the IM can be read through a Program Pointer and for the best performance, that the output of a certain address location of the IM should be at least 128-bit-wide to hold at least two 64-bit instruction packets in a dual-packet VLIW manner. It is possible to output a fewer bit width and consequently be forced to cascade the output instructions, however, this comes at the cost of degrading the CPI (Cycles Per Instruction) of the underlying implementation. It is also necessary that the IM provides a means to write program instructions, through either a manually-controlled or a cache-controlled write port. Other details of the hardware implementation are at the concern of the designer's micro-architecture choices.

6.4.1.2 The Control Unit

The CU (Control Unit) is the heart of the Sequencer and performs the most important tasks such as loading the IM with program instructions, booting the overlay and decoding and issuing different instruction streams to the different receivers. For example, C-Type instructions are handled differently than other types because the related decoded streams are executed solely by modules inside the Controller part.

6.4.1.3 Interfacing with a host

The overlay architecture can be run in a stand-alone mode, or be interfaced with a host. The CU implementation should manage either mode of deployment. In the case of a host-interfaced implementation, the CU allows the control of the overlay from within the host and exposes its GM memory space to it. Details of the implementation of either modes of operation are left to the free choices of the micro-architecture designer.

6.4.2 The Direct Memory Access Engines

A DMA (Direct Memory Access) is a moving engine for blocks of contiguous data. In fact, it can move data between a specific GM bank and the corresponding broadcast cluster's BM banks.
The DMA is a major contributor to the high efficiency of DRAGON by freeing the sequencer time when moving data back and forth between the Accelerator and the Controller.
While the detailed operation allows some micro-architecture design freedom, the general behavior is that it can accept specific configuration frames from the sequencer's Control Unit. These configuration frames instructs the DMA about the amount of data that needs to be transferred (the number of byte bursts through the AXI protocol), the direction of the transfer (either a write or a read operation), as well as the BM and GM offset addresses.
DRAGON is a SIMD architecture where every operation is executed in a parallel manner. The DMA engines are no exception. It is necessary that the micro-architecture implementation provides the base pointer for each GM region that is connected to a BC. Typically, the GM memory space is divided in different regions implemented as separate banks that are each connected to a unique BC. Besides, a special DMA engine can be used to load the program instructions from the GM into the IM.

6.5 The Accelerator

6.5.1 The Broadcast Cluster: A Modular Approach

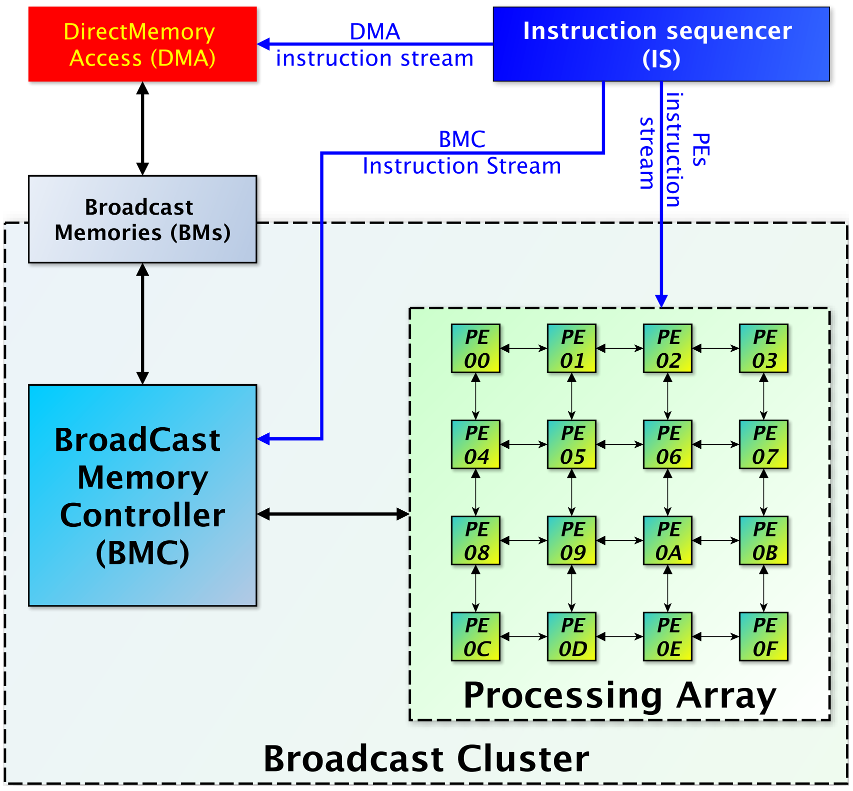


Figure 6.3: The DRAGON Broadcast Cluster general architecture.

6.5.1.1 Broadcast Memories and Broadcast Memory Controller

DRAGON adopts a modular design with different layers of different granularity. In the lowest level resides the PE which has the lowest memory level as well, the LM. The PEs are grouped in an array of 4x4 PEs, in a 2D grid structure and typically interconnected through a 2D-Mesh topology. These PEs, together with the BMs (Broadcast Memories) and the BMC (Broadcast Memory Controller) form the BC (Broadcast Cluster). This modular structure enhances the maintenance of DRAGON micro-architecture implementations and eases targeting FPGA architectures with multiple dies, by simplifying the mapping of BCs to a given die. Fig. 6.3 depicts the general architecture of the BC. In fact, a BC accepts two separate streams of instructions from the sequencer. One instruction stream is destined to each PE for execution in a SIMD manner, while the other is further decoded by the BMC, that issues subsequent control signals to the PEs and manage the flow of data between these PEs and their respective BM banks, in the BC side. The sequencer issue another instruction stream that configures the DMA to transfer data back and forth between the other side of the BM banks and the GM. Besides, the BM is implemented in a banked manner and there are sixteen banks in total in each BC, where each BM bank has a 12-bit address pointers to access them with a read or write operation.

6.5.2 The Processing Element

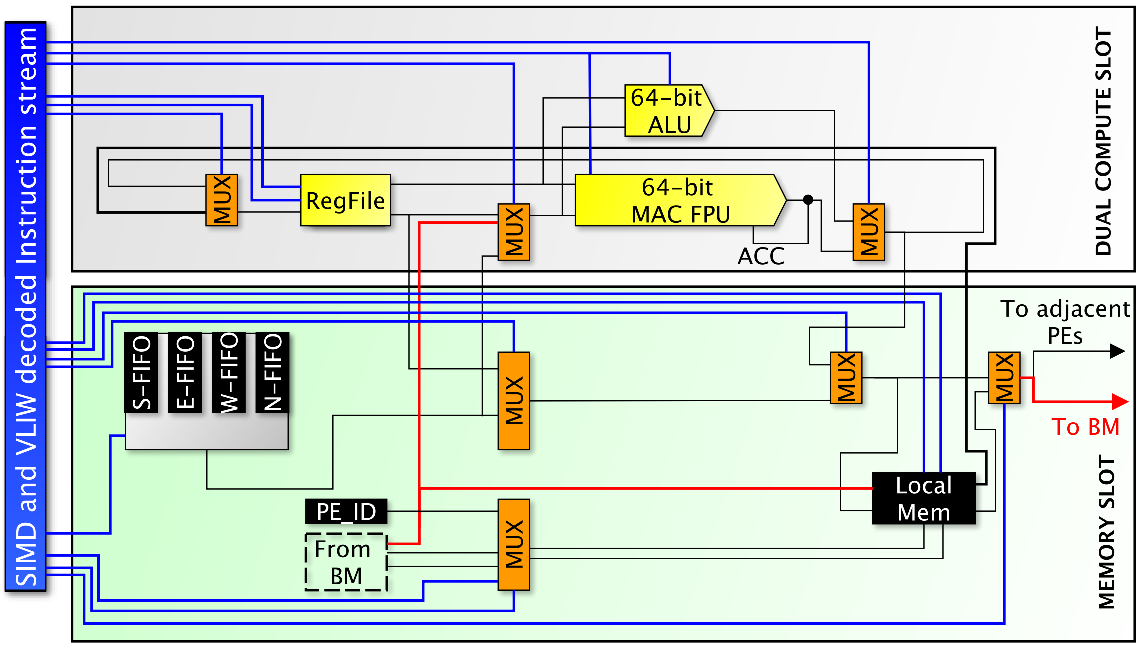


Figure 6.4: The DRAGON PE general architecture.

The DRAGON PE is expected to process the stream of VLIW instructions coming from the sequencer. Therefore, the PE matches the structure of packets in this stream. The VLIW instruction typically embeds both a computational and a memory operation instruction packets and therefore a PE is structured into slots that manage each type of instruction. These two slots are called Memory Slot and Dual Compute Slot and a typical architecture is illustrated by Fig. 6.4.

The DRAGON PE ensures the execution of the decoded and statically scheduled VLIW instructions that are split into two packets consisting of memory and computational operations using 64-bit-wide data. The DRAGON PE supports both 64-bit integer and floating-point double-precision operations as well as local data movement between register file and local memory as well as specialized neighbor communication instructions.

The number of pipeline stages is a matter of micro-architecture implementation choices, however the broadcast memory controller has to account for such number to ensure the synchronized operation with the data it broadcasts to the PE.

6.5.2.1 The Dual Compute Slot

The DCS (Dual Compute Slot) shown in Fig. 6.4, is one of the two compartments of the DRAGON PE and it ensures the execution of 64-bit integer operations using the ALU (Arithmetic and Logic Unit). It also allows computations using double-precision floating-point format, through a custom MAC FPU (Multiply-ACcumulate Floating-Point Unit). These two units share the same input operands and therefore the compute operation output must be pass through a multiplexer that selects the active unit output in a given clock cycle.

Moreover, as detailed later in the DRAGON instruction set architecture, the result of the computation is not always written back to the register file. In fact, it can be broadcasted to other PE neighbors and/or directly written to local memory, therefore, another multiplexer is required to select the input data to write in the register file destination register. Furthermore, DRAGON allows operations not only using register operands but also other inputs such as data incoming from broadcast memories, immediate values, or communication buffers that store the gathered data from the PE neighbors.

6.5.2.2 The Memory Slot

The MS (Memory Slot) implements a set of communication buffers whose pointers are managed in a circular buffer FIFO (First In First Out) -based manner.

In contrast to the DCS which performs purely computational instructions, the MS manages data movement operations such as :

- LM loads and stores
- BM loads and stores
- passing data that are stored in communication buffers to a PE neighbor
- Scattering data from either the local memory, the register file or one the outputs of either

the ALU or the FPU, to PE neighbors and gathering incoming data into the respective communication buffer.

It is possible to perform a computation, store its result into the LM while loading a datum from LM into the register file , thanks to the multiple-issue of instructions using the VLIW model. While DRAGON decodes and concurrently executes two instructions inside the DCS and MS slots, it allows more operations to be performed per clock cycle. In fact a single VLIW instruction packet can for example, provide the result of a computation through the ALU or the FPU, store the result to LM, scatter the result to neighboring PEs, gather the incoming data from these neighbors and loads a new datum from LM into the register file. This boosts the efficiency level of the PE, in particular by reducing the performance cost of data movements, by overlapping memory transfers with effective computations.

Besides, every MS contains a PE ID (Processing Element IDentifier) that allows selective data transfers from BM to LM through a masking scheme that compares this PE ID with the mask issued by the BMC after decoding the original mask contained into the MS instruction.

6.5.3 The Topology and Dimension of the Interconnect

The DRAGON overlay adopts a tightly-coupled architecture model where all PEs can exchange locally stored data through a direct interconnection network. The DRAGON interconnect is based upon a switchless topology, where each PE represents a single node that is connected to a set of adjacent neighbor nodes. The network interconnect degree, defines the number of these neighbors for each PE. The DRAGON interconnect network also adopts a buffered approach that allows storing the exchanged data between adjacent local and distant connected PEs. As such, directional communication buffers behaving in a circular buffer FIFO manner, allow each PE to store data incoming from multiple neighbors or scatter the local data to these neighbors. Depending on the implementation interconnect topology and degree, the number of neighbors may differ. However, the base DRAGON overlay allows connections with up to eight PE neighbors, as this is an architectural limitation of the underlying base instruction set architecture presented in Chapter 4. This limitation also takes into consideration the scarcity of physical wires, known as SLLs (Super Long Lines) and which connect die regions known as SLRs

(Super Logic Regions) in multi-die FPGAs [111]. For example, in the case of HBM-enabled multi-die FPGAs, these wires must be used to connect HBM banks to their respective BCs, and as such these wires may quickly become a bottleneck for implementation scalability of the DRAGON overlay, both regarding its size (number of deployed PEs) and its interconnect degree. Nevertheless, it has been shown in [4] how a micro-architecture implementation of the DRAGON overlay architecture is capable of implementing a 3D degree of interconnect or even a 4D degree, which provide six and eight connected PE neighbors for each node, respectively.

Regardless of the target FPGA architecture, and from a pure architectural perspective, each PE has to embed a specific number of data communication buffers, depending on the degree of the desired interconnect topology.

For example, a 2-D-Mesh/Torus interconnect requires four communication buffers to store the input data coming from four directions, namely, North, West, East and South PEs neighbors.

Moreover, in a 3-D-Mesh/Torus topology, two extra communication buffers are required to accommodate the two additional remote PEs connections, which elevates the total number of communication buffers to six (four from local neighbors and two from remote neighbors).

Furthermore, in a 4-D-Mesh/Torus topology, an added two extra communication buffers are required to accommodate the two additional remote PEs connections, which elevates the total number of communication buffers to eight (four from local neighbors and four from remote neighbors).

6.6 Summary

This chapter presented a general overview of the DRAGON many-core overlay architecture. The description begins by introducing the different parallel processing paradigms adopted by the proposed architecture. Then, it dives into system level details such as the architectural split between the Accelerator and the Controller, while providing a brief description of their internal structures. This chapter also provides an example of a typical processing element architecture and the way it should handle the adopted VLIW programming approach. Ultimately, this chapter serves as a basis to the subsequent micro-architecture implementations presented in Chapter 7 and Chapter 8.

Chapter 7

Baseline micro-architecture implementation of DRAGON

7.1 Introduction

This chapter presents the baseline micro-architecture implementation that serves for the base preliminary evaluation. Mainly, this chapter summarizes the work presented in [3]. As such, the micro-architecture presented in this chapter is labeled the Baseline DRAGON. Subsequently, the details of its micro-architecture implementation on the target FPGA [37] will be presented and discussed in the following sections.

7.2 Micro-architecture of the baseline DRAGON

This section presents the Baseline DRAGON micro-architecture implementation details. Here, Fig. 7.1 illustrates an overview of such an implementation, in which it depicts two major parts that operate in tandem. The first major part is the Controller that orchestrates overall operation by decoding and issuing SIMD instruction streams, through its sequencer module. The Controller, implements as well DMA engines that ensure programmed data transfers between the GM banks and their respective BMs in the Accelerator part which is the core computing part of DRAGON. This Accelerator consists of multiple duplicated clusters that contain each an array of 16 PEs, 16 BM banks, and a BMC (Broadcast Memory Controller) that acts in turn as a partially local DMA between BMs and each PE's local memory. The PEs perform the execution

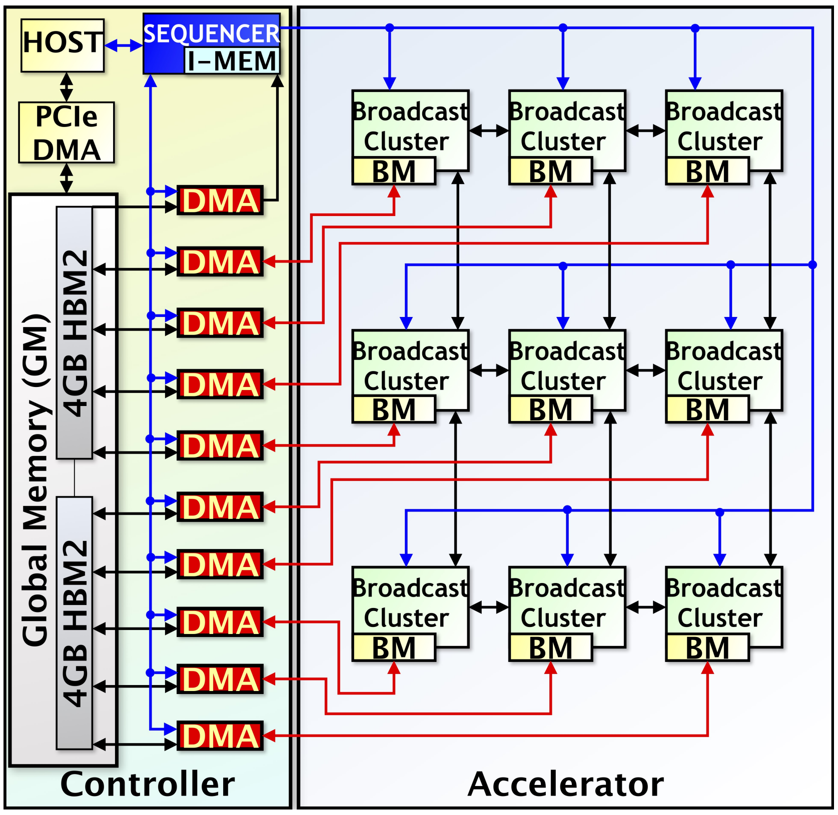


Figure 7.1: Micro-architecture of the Baseline DRAGON overlay [3, 4].

of the SIMD instruction stream provided by the the Sequencer. The BMs are in fact intermediate buffers between GM banks and LMs, and are connected through DMAs in the Controller side, and BMC in the Accelerator side. The program instructions are loaded from their dedicated GM bank into the IM (Instruction Memory), through a dedicated DMA engine. The host uses the PCIe infrastructure provided by the FPGA vendor (Xilinx) RTL kernel model to move data back and forth between the FPGA device and its host computer.

7.3 The Sequencer

The Sequencer is the key component of the Controller part. Fig. 7.2 illustrates the main modules of this component. Among these, it shows the IM that stores program instructions, the AXI (Advanced eXtensible Interface) Lite Control Interface that interfaces the FPGA device with the host through the exchange of configuration parameters and interrupt signals, and finally the CU (Control Unit) that plays an orchestrating role for all of the operations inside the overlay, including the central task of reading/decoding program instructions from the IM and issuing multiple SIMD decoded instruction streams towards PEs, DMAs and BMCs.

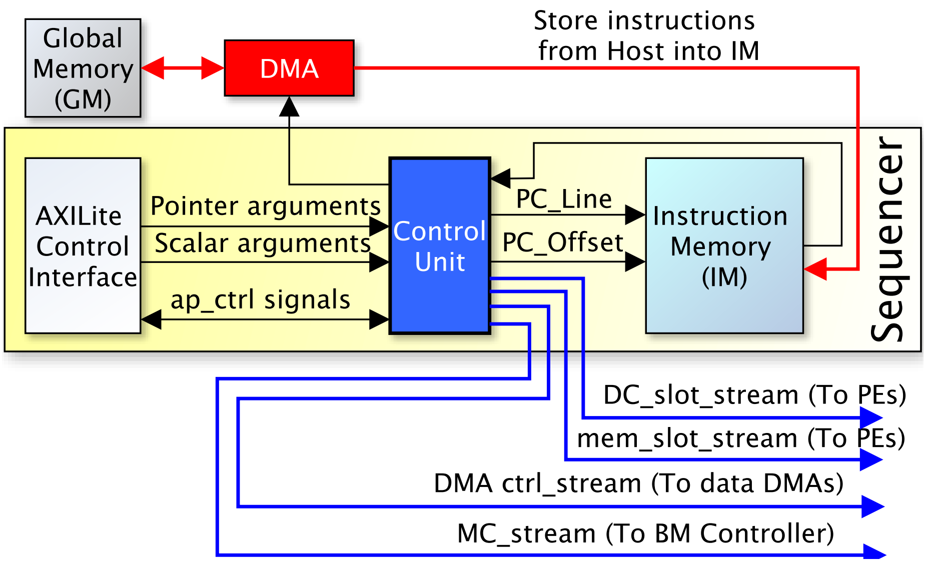


Figure 7.2: Micro-architecture of the Sequencer [3].

The host-side call of the OpenCL prototype function that abstracts the DRAGON overlay sets a special register bit in the AXI Lite Control Interface. When, this bit is set, a start signal is generated and instructs the CU to start the overlay operation by either entering into a boot sequence that loads new program instructions from their GM bank into the IM, or by jumping directly into the normal operation mode where reloading program instructions is not required and the same program is re-used instead. The selection of either modes, depends on the configuration parameter set by the user in the host OpenCL overlay abstraction function. In case of the first mode of operation (boot sequence), another required parameter indicates the memory size of the program (in Bytes). After the completion of the boot sequence, the overlay moves to normal operation state. In this state, the CU manages the reading of program instructions from the IM by updating the value of the PC (Program Counter). The read instructions are then decoded and new control signal streams are issued, at every clock cycle, towards their corresponding recipients. In fact, a total of three streams are generated. The first is a VLIW stream containing two packets that are sent for execution on the MS and DCS slots of each PE, respectively. The second is a configuration frame used to control the operation of the DMA engines such as read or write mode and the amount of data to be moved. The third is sent to the BMC to synchronize its operation with the PEs, change the broadcasting mode, and control the flow of data between BMs and LMs.

The special C-Type control instructions are solely executed in the Controller part. For example, these instructions manage program loops where seven levels of nesting are supported in the Baseline DRAGON micro-architecture implementation. The completion of data processing is detected when an instruction having the STOP opcode is encountered. This should indicate that all program instructions have been executed and all the processed data was written into their corresponding GM banks. In this situation, an interrupt signal is generated which sets a dedicated register bit in the AXI Lite Control Interface, notifying the host that the processed data is ready to be copied to their corresponding host buffers.

7.4 The Instruction Memory

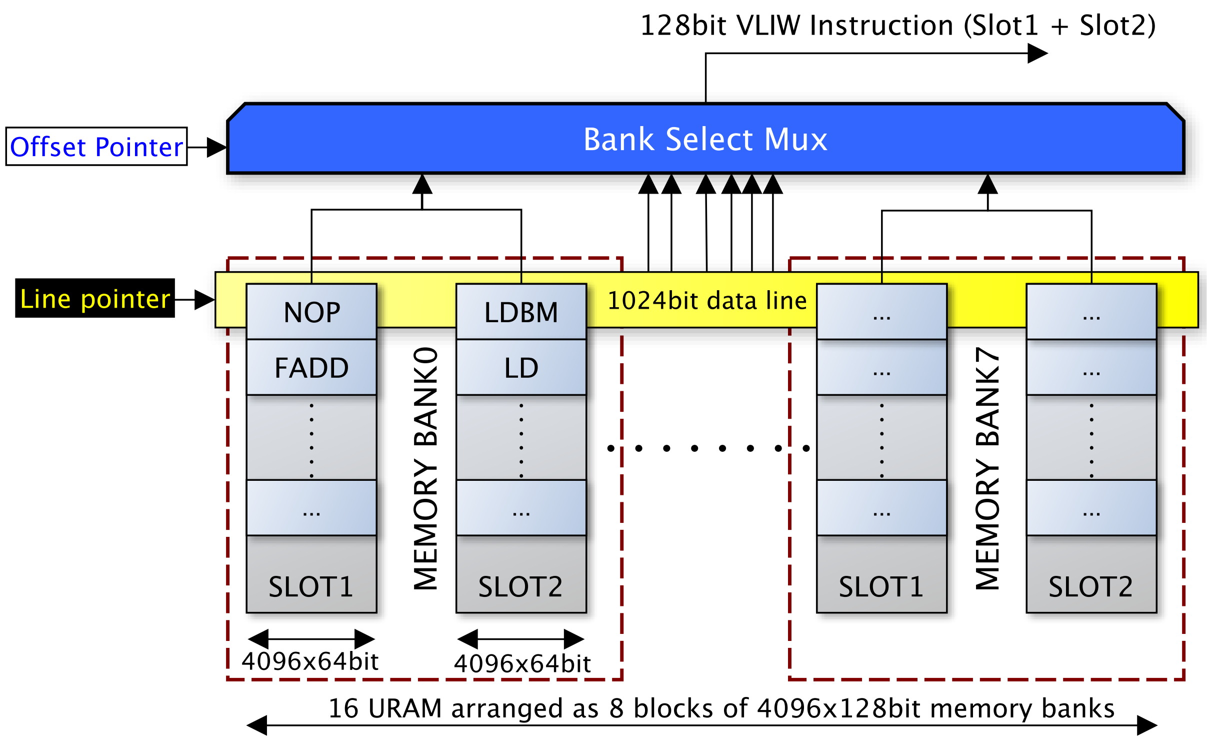


Figure 7.3: Micro-architecture of the Instruction Memory [3].

The IM allows the storage of program instructions that are loaded from the dedicated GM bank during the boot sequence. It is implemented using the largest capacity on-chip memory resource on the FPGA that is the URAM memory. It offers 512 KiB of storage thanks to the use of 16 physical URAM memory blocks. These are logically arranged into eight blocks that are each providing 128 bit output and allowing the storage of 4096 128-bit VLIW instructions (2 x 64-bit slots). An example of such an implementation is illustrated by Fig. 7.3.

In this implementation, a memory row contains 8 128-bit VLIW instructions (a total of 1024 bits). The CU reads instructions by generating a PC that allows to point to the equivalent IM location address. This PC consists of two parts, the LP (Line Pointer) that indicates the row in which the instruction resides and the OP (Offset Pointer) that indicates which of the 8 logic partitions contains that instruction, and is used as a selector to the 8x128-bit input multiplexer. The data on the output of this multiplexer is subsequently sent to the CU for decoding.

Nonetheless, this physical arrangement of the URAM memories was selected to maximize the storage bandwidth by matching the implemented AXI data bus width (1024-bit data bus). In fact, these memories are implemented in a dual-port mode which provides in total 1024-bit inputs arranged as 16 64-bit data input ports connected directly to the 1024-bit AXI data bus. An equal number of 64-bit data output ports is also connected to the bank select Mux (multiplexer) shown in Fig. 7.3. This bandwidth maximization allows shortening the boot sequence program load time during which the GM dedicated program bank (HBM bank on the FPGA) is active. As such, it effectively reduces the related energy consumption.

7.5 The Broadcast Cluster

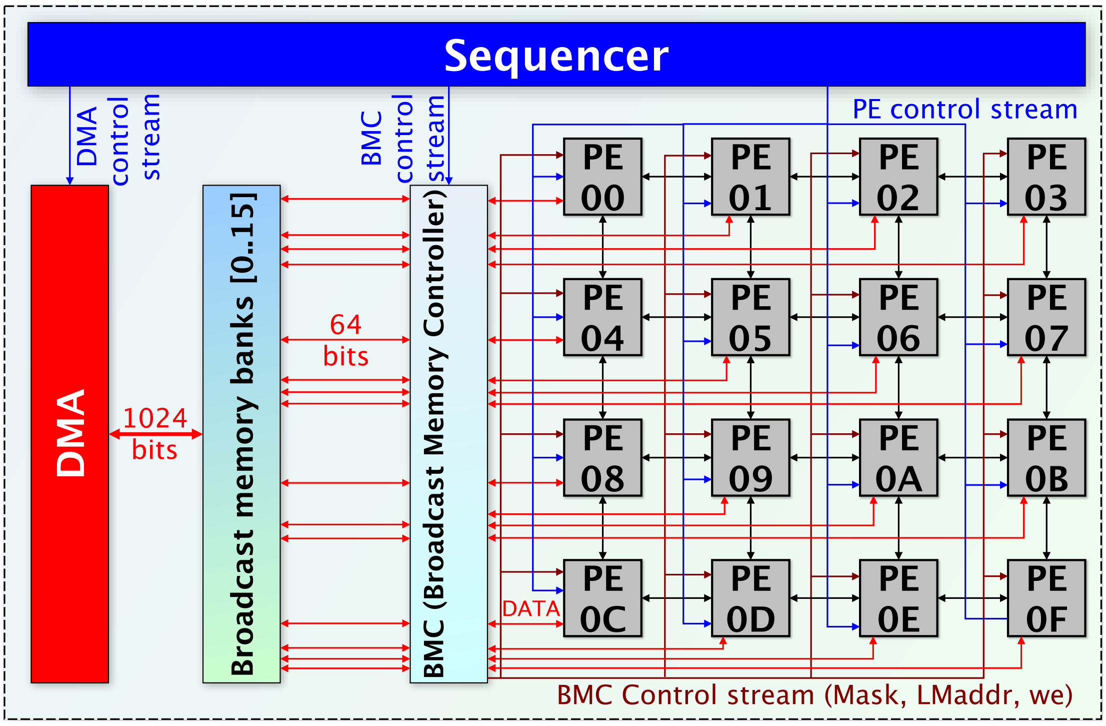


Figure 7.4: Micro-architecture of the Broadcast Cluster [3].

The parallel nature of DRAGON is manifested through different aspects. First the use of a banked GM implementation which allows a fast parallel-access to data either during write or read operation. Then through the modular implementation of relatively small clusters (BCs) that embed each an array of 4x4 tightly-connected SIMD operating PEs. Ultimately, through the splitting of the operation of these PEs by adopting a VLIW model where two slots may perform different kinds of operations (data movements and pure computations). These PEs are 2D-Mesh interconnected and each can exchange its local memory data with the respective BM bank through the BMC. The BM acts as an intermediate buffer that links between LMs and GM banks. To reduce the eventual energy overhead caused by data movements, it is necessary to maximize the bandwidth of BMs while ensuring a minimal on-chip memory resource utilization, to reduce the area size and the related power dissipation. As such, every BC implements a total of 16 URAMs, one for each BM, so that each PE is connected with a respective BM from one side, while the other BM side is connected to the DMA-BM AXI-based 1024-bit data bus (16 x 64-bit), as depicted by Fig. 7.4.This allows separate concurrent accesses to BM banks from both Accelerator and Controller sides (PE-side and GM-side), which facilitates a double buffering pipelined approach to move the data more efficiently, in either directions. For an even enhanced efficiency, the BMC is able to act as a half duplex DMA that manages bulk data transfers from BM to LM following a call to an LDBM instruction.

7.6 The Broadcasting feature implementation

The base DRAGON ISA offers the possibility of broadcasting a single data from any BM bank to all the PEs inside a BC. This data broadcasting may be useful to reduce bandwidth requirements in multiple applications such as vector-matrix multiplication. The DRAGON ISA also offers the possibility of extension towards other schemes of data broadcasting that can be used in other specific applications. Moreover, this broadcasting feature allows the access to the combined full memory space of all the BM banks, when loading data into a specific LM. For example, when broadcasting is not required (not selected by the programmer), each BM bank outputs data solely to its respective LM with the same PE ID (Processing Element IDentifier). On the other hand, when the broadcast feature is selected, any single BM bank output, among all the 16

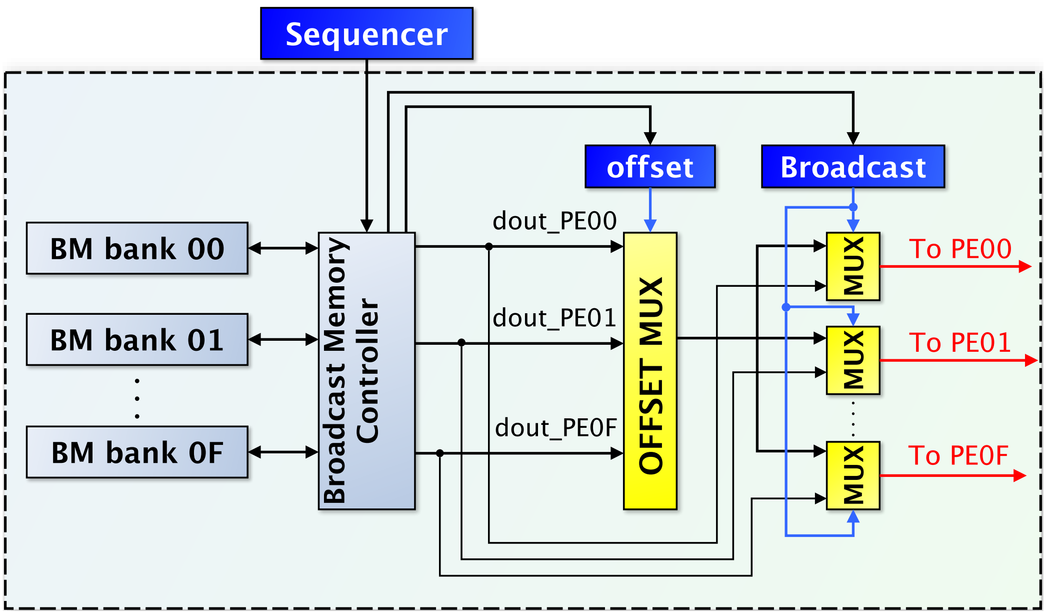


Figure 7.5: Micro-architecture implementation of the two-level broadcasting feature.

existing banks, may be broadcasted to each PE in the BC. This is extremely useful for it helps reduce the memory bandwidth requirement by 16 times. This implies, it can reduce the time to move shared data into each local memory by 16 times as well, resulting in a significant decrease in energy consumption. Fig. 7.5 illustrates a typical micro-architecture implementation of this feature through a two stage multiplexing logic. The first stage selects a single BM bank output among the 16 available banks through an offset selector that is obtained from the instruction field "BrOffset" in Fig. 4.5. A second multiplexing stage is implemented using 16 2-input multiplexers. When the Broadcast feature is set in a program instruction, the output from the first stage multiplexer is broadcasted to all PEs of the same BC. Otherwise, every BM bank output is sent solely to the respective PE with the corresponding PE ID.

7.7 The Processing Element

The PE architecture and micro-architecture are key elements in the quest towards an energy-efficient FPGA-based many-core overlay. The Baseline DRAGON PE, depicted by Fig. 7.6, adopts an implementation with a total of seven pipeline stages. Pipelining is a critical ingredient for performance as it allows temporal parallelism, in addition to boosting the operating clock speed by reducing the logic levels and shortening critical paths. The logic levels reduction not only improves the computational performance but it contributes as well to the minimization of

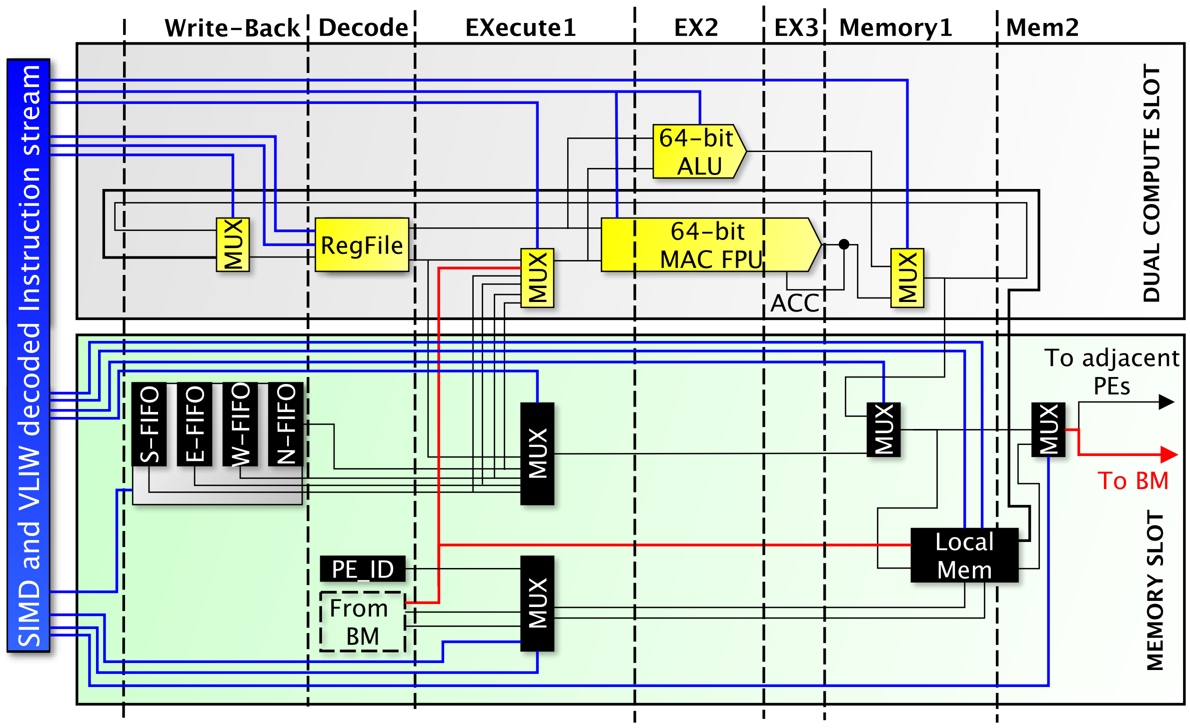


Figure 7.6: Micro-architecture of the Processing Element [3].

glitches that are responsible for a large part of the power dissipation in digital CMOS circuits. Moreover the Baseline DRAGON PE provides two slots (Dual Compute Slot and Memory Slot) that ensure the concurrent execution of the decoded VLIW packets, which results in an efficient overlapping of data movements such as memory loads and stores with the effective computations on the DCS slot. For example, A PE is capable of performing a computation through its FPU, scatter the result to neighboring PEs, while loading new data from BM to LM or from LM to the Register File.

These computation are performed with the help of the 64-bit integer ALU and the double-precision MAC FPU. The RTL-based description of the MAC FPU, whose details are illustrated by Fig. 7.7, uses pragmas to ensure the utilization of DSP modules instead of distributed logic (LUTs) for the multiplication operation which results in a more compact area that leads in turn to a reduced power dissipation. In addition, the compact implementation leads as well to a better routing quality which translates to better timing as well as better computational performance outcome. Besides, the LM is implemented using the URAM memory resource which provides a 12-bit addressable space for 4096 64-bit different locations. This LM is capable of concurrent load and store to/from the Register File. Furthermore, the large capacity Register File provides

256 64-bit locations, which leads to the reduction of unnecessary data movements to/from the LM, and results in a reduced overall energy consumption.

Nevertheless, the low-latency single-cycle multiplication operation is a limiting factor for clock speed while neighbor communication buffers implemented in separate BRAMs are a limiting factor for scalability and for increasing the degree of interconnect beyond 2D. These limitations will be further discussed in the experimental evaluation chapter, while the key improvements will be presented in detail, through the enhanced micro-architectures in the subsequent chapter.

7.8 The Multiply-ACcumulate Floating-Point-Unit

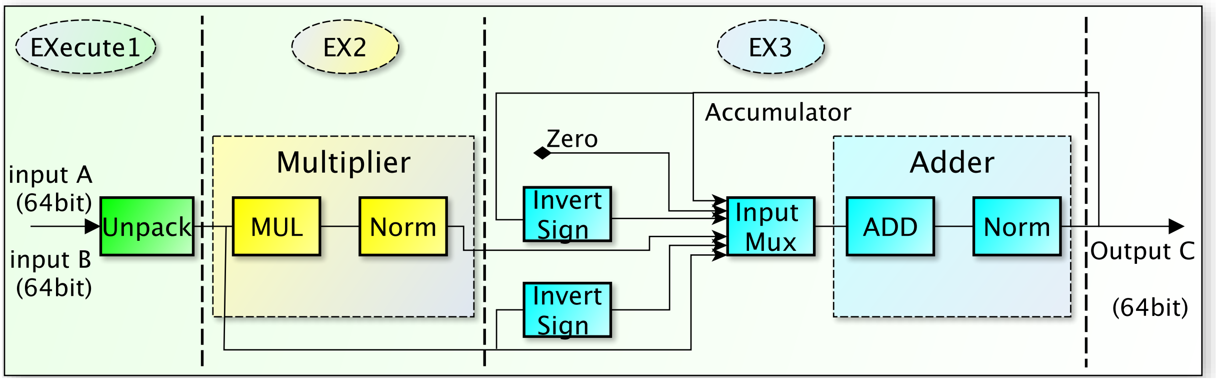


Figure 7.7: Micro-architecture of the Multiply-ACcumulate Floating-Point-Unit (MAC FPU) [3].

The MAC FPU in the Baseline DRAGON overlay is implemented as a cascade of a single-cycle multiplier followed by a single-cycle adder, both performing operations on double-precision floating-point data operands. The goal is to provide a low-latency unit that mainly supports addition, subtraction and accumulation through a call to fused multiply-accumulate instructions. In this implementation, the MAC FPU adopts truncation as the default rounding mode, to simplify the required computation by resulting in fewer logical levels. This results in turn in a minimal resource utilization and consequently a minimal power dissipation.

The MAC FPU adds a multiplexer and extra logic that performs some transformations on the input operands of the adder part. For example, it can invert the sign of the second operand to perform a subtraction operation, either on the pre-multiplication inputs or on the accumulated result. Besides, to uniquely perform a multiplication operation, the multiplication result is forwarded to the second operand input of the adder whereas the first operand takes zero as an input.

7.9 Summary

This chapter introduced the Baseline micro-architecture implementation of the DRAGON overlay architecture on the target FPGA in the Alveo U280 acceleration board[37]. This chapter provided the micro-architecture details for the sequencer, the Instruction Memory, the Broadcast Cluster and its broadcasting feature, the Processing Element and the Multiply-Accumulate Floating-Point Unit. The related implementation results will be presented in detail in Chapter 9.

Chapter 8

Enhancing the energy-efficiency through DRAGON2 and DRAGON2-CB micro-architecture implementations

8.1 Introduction

The DRAGON2 micro-architecture adopts the same system level architecture of the Baseline DRAGON overlay that is depicted by 7.1. However, DRAGON2 is an enhanced micro-architecture implementation of DRAGON that is heavily optimized for the target FPGA [37]. As such, it benefits from layout-aware floorplan and fine-tuned placement. The AXI-based interconnect that links the DMA engines to the GM has been improved following a mathematical formulation that was established to strike the best balance between scalability and AXI data bus width, taking into consideration the scarcity of the wires between the different regions of the target FPGA. Moreover, the communication buffers are improved through the adoption of a new scheme named compact buffering which is implemented in the compact buffering version of DRAGON named DRAGON2-CB. In addition, the clock speed has been nearly doubled by increasing the number of pipeline stages in each of the MAC FPU and the PE micro-architectures. Besides, an additional registering of signals across the FPGA regions has been added and accounted for in the PE implementation by adding a compensation stage as a second write-back stage. Finally, all these enhancements allowed scaling the overlay in size by increasing the number of deployed PEs which is doubled as compared to the Baseline DRAGON. Furthermore, the scalability of the overlay in terms of degree of the interconnect has also been improved and made DRAGON2 and DRAGON2-CB able to adopt 3D and 4D dimensions of connectivity alongside hosting larger amounts of the improved PEs.

8.2 Micro-architecture Enhancements

8.2.1 Optimizing the GM-BM AXI-based data bus interface

The Baseline DRAGON overlay implemented a 1024-bit AXI data bus that connects each GM HBM2 bank to a set of 16 BM banks in each broadcast cluster. As a result the 1024 bits can be exactly divided into 16 64-bit data, and the BM banks can be written to, simultaneously, at each AXI transaction beat. The simplicity of such an implementation came at the cost of an increased routing difficulty that limited the overall scalability, because it was bottle-necked by the scarce wires connecting the different dies inside the FPGA. In contrast, the Broadcast Cluster in the DRAGON2 and DRAGON2-CB overlay implementations, reduces the size of the AXI-data bus to just 256 bits, while introducing a multiplexing control logic to match the original behavior. Furthermore, the AXI-data bus has been pipelined to outperform the clock speed obtained in the Baseline DRAGON overlay implementation.

8.2.2 Improving the MAC FPU through a deeper pipeline

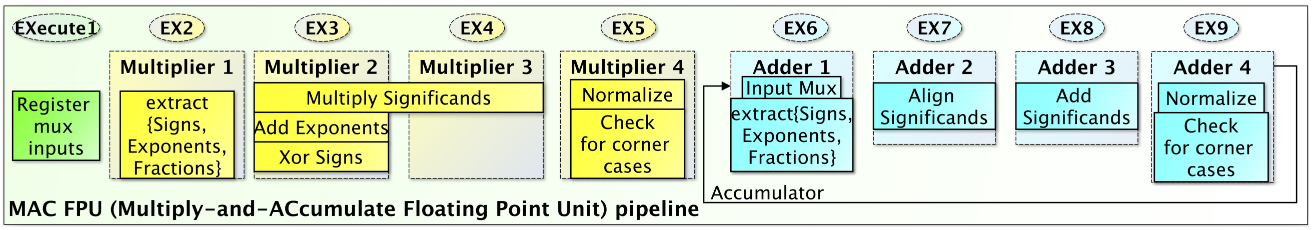


Figure 8.1: The micro-architecture of the MAC FPU of the implemented DRAGON2 and DRAGON2-CB overlays [4].

The Baseline DRAGON micro-architecture implementation suffered multiple performance and energy efficiency bottleneck. In the baseline implementation, the MAC FPU provided support for a single-clock-cycle multiplication and a single-clock-cycle accumulation operations. Despite this low-latency advantage, the resulting computational performance remained limited, because of the multiple logic levels required by these computations, which limited the achievable operating clock speed. Furthermore, these multiple logic levels increased the probability of spurious transition activity known as glitches, which leads to increased power dissipation and energy consumption.

To enhance the MAC FPU implementation, its pipeline depth has been extended to a total of four stages for the multiplication and another four stages for the accumulation operations. As such, the number of logic levels at every pipeline stage has been reduced significantly, which led to shorter paths that in turn allowed it to achieve higher operating clock speed as well a reduction in power dissipation related to glitches.

The enhanced MAC FPU detailed pipeline operations are illustrated by Fig. 8.1. The default rounding mode re-uses truncation that was previously implemented in [3], to reduce the number of logic levels and related hardware resources, resulting in a more compact implementation.

8.2.3 Enhancing the PE through extended pipeline and optimized buffering

8.2.3.1 The DRAGON2 PE

The enhanced PE micro-architecture of the DRAGON2 overlay is depicted by Fig. 8.2. The introduced enhancement consists of extending the number of pipeline stages for a higher operating speed and reduced dynamic glitch power as a result of shorter paths and decreased logic levels between stages.

In this micro-architecture, an extra (MEMory3) pipeline stage was added near the local memory output that is destined to drive data towards the broadcast memory or the neighboring PEs.

While not shown in Fig. 8.2, the data path connecting the output of a PE to the input of an adjacent one was shortened through the insertion of a pipeline register, providing more freedom to the routing tool and resulting in better overall speed and reduced capacitive load power related to the length of the underlying wires.

To maintain the synchronization between PEs, an additional write-back stage was inserted, as a compensation for the introduced delay on neighbor-incoming data signals, that is a result of the

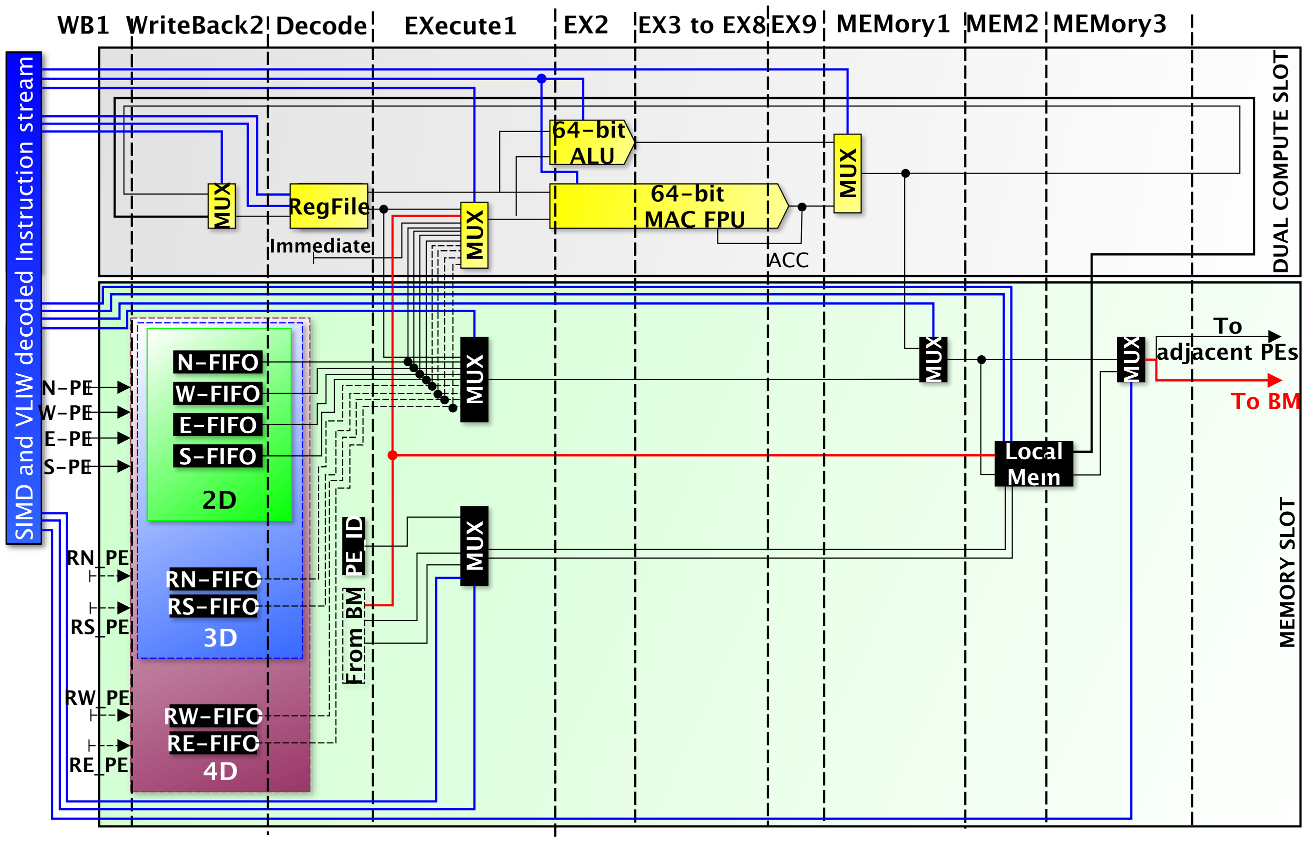


Figure 8.2: The micro-architecture of the Processing Element of the implemented DRAGON2 overlay [4].

extra pipeline register described earlier.

Besides, the ALU operation has been extended through an additional pipeline stage. This allowed the integer multiplication to be implemented with an additional clock-cycle, resulting in an improved operational speed and reduced glitching power.

In total, the new PE micro-architecture in the DRAGON2 overlay embeds 15 stages as compared to just 7 in the baseline version, taking into consideration the extended pipeline of the MAC FPU (8 stages).

8.2.3.2 The DRAGON2-CB PE (Compact Buffering PE)

Fig. 8.3 illustrates the new micro-architecture of the DRAGON2-CB PE which benefits from the Compact Buffering (CB) as compared to the DRAGON2 PE.

While seemingly abundant in large FPGA devices, the amount of hardware resources remains relatively a limitation for scalability. In fact, different kinds of hardware resources may be available in different amounts. It is therefore important to balance the utilization of the different kinds of

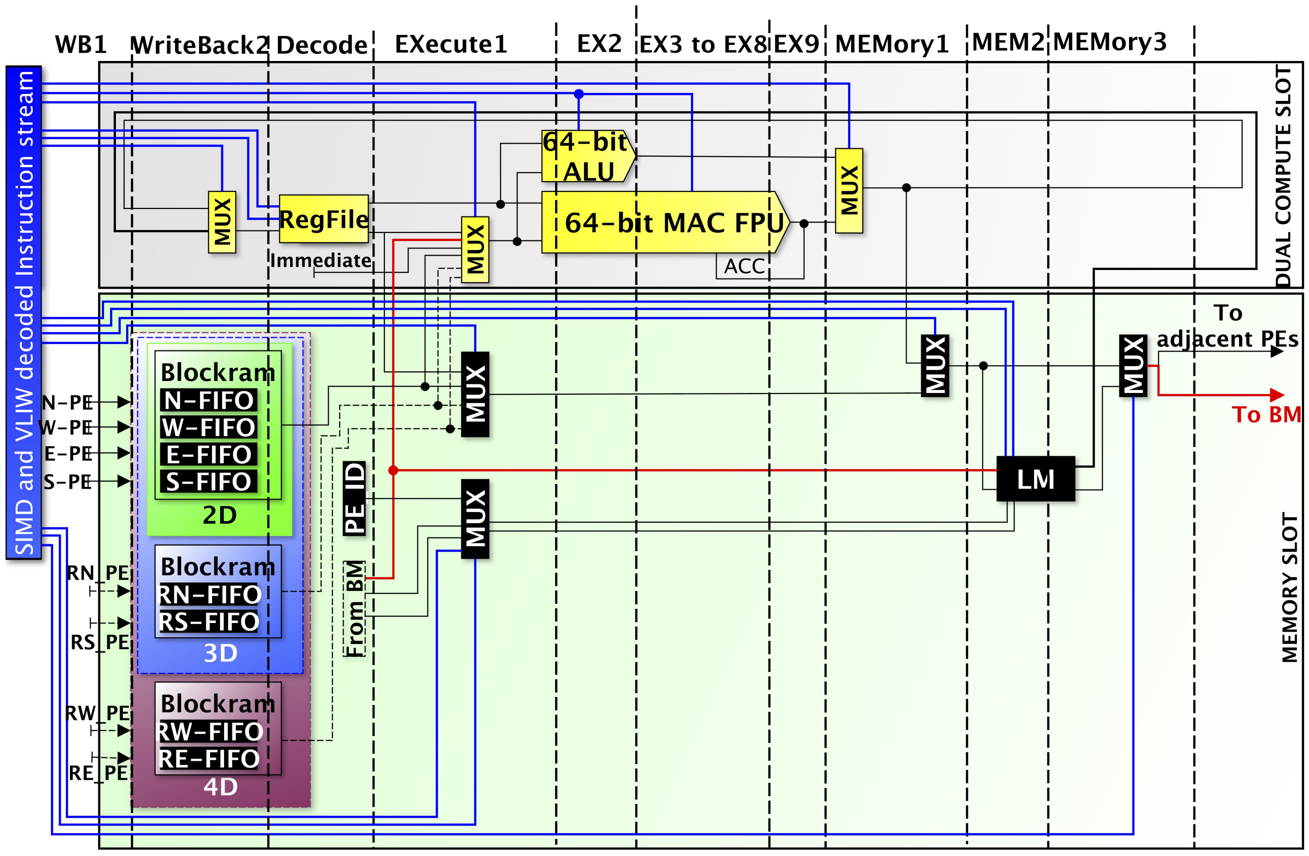


Figure 8.3: The micro-architecture of the Processing Element of the implemented DRAGON2-CB overlay [4].

resources with regards to their availability on a target device. In particular, the DRAGON2 PE implements communication buffers as separate BRAMs and uses 4,6 and 8 BRAMs for designs with 2,3 and 4D interconnects, respectively. Clearly, this implementation choice may become a bottleneck for scalability (deploying more PEs), especially in a higher degree of interconnect.

To solve this issue, the DRAGON2-CB PE, shown in Fig. 8.3, maintains most of the aspects of the DRAGON2 PE while it implements an enhanced buffering scheme called Compact Buffering, that will be explained later. In this scheme, the DRAGON2-CB uses 1, 2 and 3 BRAMs for designs with 2,3 and 4D interconnects, respectively, resulting in a significant decrease of on-chip memory utilization, reduced power and nearly similar performance levels as will be shown in the experimental evaluation chapter.

8.2.4 The Compact Buffering Scheme

8.2.4.1 A concept overview

The compact buffering scheme aims at reducing the number of BRAMs used by communication buffers to exchange the data between neighboring PEs. This reduction aims to maintain a good scalability of the overlay without compromising the computational performance.

The compact buffering technique consists of grouping multiple communication buffers into a single BRAMs and efficiently scaling the used amount of BRAMs along the increase in the dimension of interconnect, to maintain an acceptable bandwidth for write operations, instead of implementing each communication buffer into its own BRAM. Since the proposed scheme is targeting a software-programmable many-core overlay architecture, parallel write operations to a single BRAM port should be handled by the user through the DRAGON ISA software instructions by deferring concurrent writes to an ulterior clock cycle while saving the corresponding data into the Local Memory that provides a relatively large capacity backup storage.

Fig. 8.4 shows the proposed scheme for 2D, 3D and 4D interconnects, along with the control logic for write operation and an example of memory space mapping for each communication buffer dedicated for each direction of data transfer with neighboring PEs.

An interesting approach to implement multi-channel FIFOs in a single BRAM has been proposed in [114], however the mere adaptation by allocating memory space of all communication buffers into a single BRAM is simply not efficient enough to maintain sufficient bandwidth for a high computational performance outcome, in particular with 3D and 4D interconnects. In fact, the work in [114] proposes the utilization of registers to buffer parallel incoming data which are then written in-order in a sequential manner. In contrast, the proposed compact buffering allows more flexibility by using the Local Memory of the PE as a larger backup storage while the DRAGON ISA provides the necessary software support to handle write operations in the order chosen by the programmer. Furthermore, the target sub-buffer write address of BRAMs is obtained directly from the multiplexed write pointers and do not require the addition of input channel information as these pointers are incremented within the address range of the allocated respective sub-buffers. Moreover, the use of a full FIFO for communication buffers is over-provisioned as the software implementation may remove the need to check status flags and thus simplify the hardware implementation. As a result, communication buffers have been implemented as circular buffers with a simplified control scheme that removes the status flag checks and the underlying logic resources. While in a 2D interconnect, all circular buffers were implemented in a single BRAM, higher dimensions of interconnect gradually increase the number of BRAMs to maintain a high computational performance when compared to a regular buffering scheme as will be shown in the experimental study related to DRAGON2-CB in Chapter 9. Consequently, the compact buffering approach reduces the required number of BRAMs from 4, 6 or 8 to 1, 2 or 3 in 2D, 3D or 4D interconnects, respectively.

8.2.4.2 Read/Write ports adaptation

A BRAM in SDP (Simple Dual Port) mode [115], has two ports, one for reading and another for writing 64-bit data. The write port consists of three inputs, namely, a write enable (wren), a 64-bit write data input (wrdata) and a write address input (wrADDr). These inputs are depicted by Fig. 8.4. The mapping of more than one FIFO-based buffer in a single BRAM requires external multiplexing and control logic to manage the write and read operations. The work in [114] provides a general control scheme for mapping multi-channel FIFO into a single BRAM. Nonetheless, the use of circular buffers in this thesis instead of full FIFOs, simplifies the overall control logic for example by removing unnecessary hardware for checking status flags. A write to a given sub-buffer is instructed by software instructions. The decoded instruction issues an 8-bit write signal where each bit is mapped to a write enable for a specific communication buffer. The DRAGON ISA description provides the necessary details on this mapping. The write operation also selects which data input from adjacent PEs is forwarded to the data input of each BRAM and which corresponding write pointer is forwarded to its write address. After every write request, the corresponding write pointer for each communication buffer is automatically incremented until it hits the maximum allowed value dictated by the allocated memory space, where it is reset to the initial location within the allocated address range of the corresponding sub-buffer into the BRAM.

The read scheme from communication buffers follows a similar control approach, however no enable signal is required. In fact, only one input (data address) and one output (data output) are required. The control management of read pointers is similar to that of write pointers.In

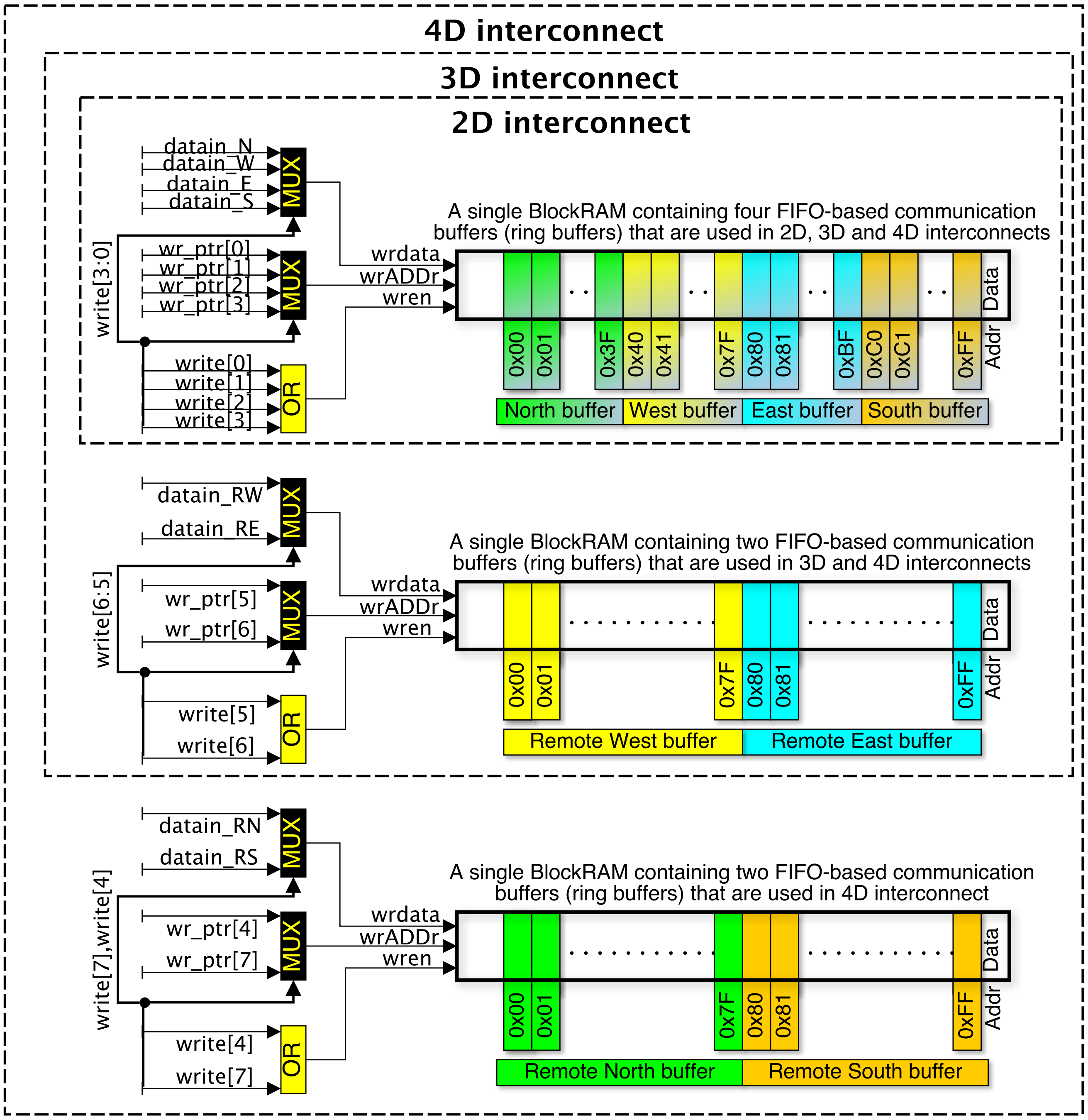


Figure 8.4: Example of compact buffering implementation on the DRAGON2-CB overlay with 2D, 3D and 4D interconnects.

other words, these pointers are incremented automatically after each read request and mapped to the address input of a BRAM read port through a multiplexer controlled by a read signal provided by the decoded instruction that requires reading data from a given communication buffer.

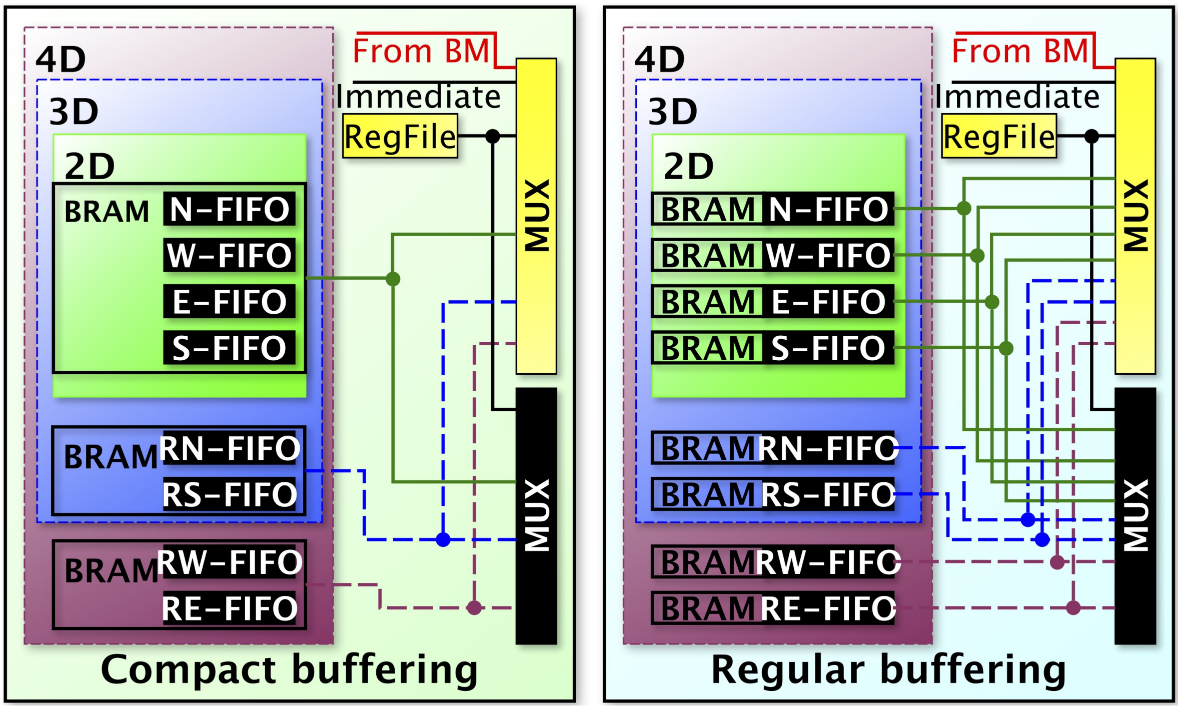


Figure 8.5: The impact of compact buffering (DRAGON2-CB) versus regular buffering (DRAGON2) on the PE multiplexing logic and the required amount of BRAM-based neighbor-communication buffers [4].

8.2.4.3 Impact on the PE micro-architecture

Using a 2D-Mesh interconnect, the PE can exchange data with its neighbors, in four directions, namely, North (N), South (S), East (E) and West (W). In the Baseline DRAGON [3] and in the enhanced DRAGON2 [4], a total of four BRAMs are used, where each uniquely implements a communication buffer for a given direction. The data outputs of these four BRAMs require a 64-bit five-input multiplexer (one input from RegFile + four from BRAMs in the MS part of the PE) as well as a 64-bit seven-input multiplexer (the output of this multiplexer drives the second operand of the ALU and FPU, in the DCS slot of the processing element), as depicted by the right side of Fig. 8.5.

Here, the use of fewer BRAMs, thanks to the compact buffering scheme, reduces the number of inputs of these multiplexers, resulting in a simplified hardware implementation as illustrated in the left side of Fig. 8.5 which also depicts similar benefits when extending the interconnect dimension to 3D or 4D.

8.3 FPGA-related Optimizations

8.3.1 Enhancing design speed through deeper pipelining

The operating clock speed plays a central role in the achievable computational performance. To reach the highest levels of circuit speed, it is mandatory to shorten signal delays and reduce logical levels by breaking further the circuit critical paths through additional pipeline registers. In particular, it is mandatory to consider the technology related to the target FPGA, including the relative location of its HBM memory banks as well as its multi-die nature.

For the best quality of results, it is necessary to apply deeper pipelining at the RTL level. This can be enhanced further by guiding the FPGA vendor synthesis and implementation tools through pragmas introduced in the HDL sources and strategies adopted during implementation phase.

A static timing analysis was conducted on the Baseline DRAGON overlay and has shown multiple targets for improving overall design speed on the target HBM-enabled multi-die FPGA. Among these, the large-width AXI data bus connecting the BM with the GM, the non-registered PE-to-PE links as well as the high-fanout instruction stream that cross the SLR boundaries and lead to a limited operating clock speed.

- **Pipelining the GM-BM AXI bus** In the target FPGA [37], there are three SLR regions, where only the bottom region (SLR0) contains HBM memory banks as shown in Fig. 8.6. That means the longest and most critical path will connect one HBM memory bank to a BC that resides on the top SLR region (SLR2), unavoidably crossing two SLR-to-SLR boundaries. The Baseline DRAGON overlay did not introduce any pipeline to the AXI data bus connecting HBM banks with the BM of each BC and therefore it resulted in a limited operating clock speed and relatively low performance and power efficiency of the overall implementation. The AXI protocol [116] prohibits the use of combinatorial paths between input signals and output signals. This introduced an increased level of complexity when pipelining the underlying data bus for the AXI read or write channels. To comply with the specification, it is necessary to decouple the ready/valid handshake interface sides by inserting pipeline registers that allow back-to-back data transfers without the insertion of a combinatorial path. This can be achieved thanks to the use of a skid buffer, also called Carloni buffer [117] which can be implemented through the use of registers to perform the function of a tiny memory-footprint FIFO that contains two unique locations for data storage.

For the longest AXI-based control and data paths going all the way from HBM banks on SLR0 to BCs on SLR2, six pipeline stages have been inserted to break them further and allow better timing results. The corresponding pipeline skid buffers have been placed as follows: One close to each HBM2 bank, one close to each BC and two around each of the two SLR-to-SLR crossings. All other AXI-based paths that do partially cross the SLR boundaries or do not cross them at all, benefit as well from the same number of pipeline stages to keep the synchronization between all the AXI-based paths.

- **Pipelining the PE-to-PE interconnection for optimized SLR crossing** When two adjacent PEs are located in two different SLR regions the data signals connecting them must cross an SLR boundary region. In the Baseline DRAGON, this connection was direct, that is, there were no registers to pipeline the underlying critical paths.

  The DRAGON2 (and DRAGON2-CB) micro-architecture solves this issue by inserting a pipeline stage that breaks these paths further, shortens the related wire delays and provides more flexibility for the place-and-route tool. To compensate for the extra clock cycle needed to transfer the data between PEs, an extra write-back stage has been added to each PE as illustrated in Fig. 8.2 and Fig. 8.3.

- **Pipelining the high-fanout instruction stream bus** The decoded instruction streams sent from the Controller to all PEs as well as to all BMCs (Broadcast Memory Controllers) are high-fanout signals that complicate routing and increase the power dissipation. Worse yet, these signals unavoidably cross the SLR boundaries which introduces an increased complexity in routing them to their destination.

  To alleviate these issues, manual replication of these signals has been introduced to reduce their fanout, and multiple pipeline stages have been introduced to reduce the impact on wire delays when crossing SLR boundaries. Both enhancements have been introduced at the RTL level, leading to an improved routability which reflected positively on the resulting timing outcome.

8.3.2 Layout-aware floorplanning

An optimal routing of FPGA resources leads to the best outcome in terms of speed, area and power and is consequently a critical step in the design flow. Routing depends heavily on prior placement of resources. In fact, optimal placement leads to optimal routing and a sub-optimal placement complicates the routing step and increases path delays which in turn increases power dissipation and degrades timing.

- **Guiding placement using Relative Location pragmas** To guide the placement tool towards achieving good quality of results, Relative LOCation (RLOC) pragmas have been inserted into HDL sources to map specific modules or signals to the desired location in the chip. These pragmas instruct the placement tool on the relative location of PEs, BCs as well as AXI-related skid buffers, using a a 2D virtual grid with X and Y positions.

- **Guiding placement using pre-optimisation step script** The RLOC pragmas assist the placement tool into positioning blocks and modules relatively to each others. Subsequently, this should be followed by additional information regarding the physical placement on the actual hardware. To achieve this increased level of control over the placement outcome, a guiding script is added to the Vivado tool during the pre-opt design step. In fact, this script aims at maintaining and passing placement information, prior to the optimisation of the design netlist by Vivado.

  The placement guiding script benefits from the Vitis-established bounded regions called "pblocks" and uses RegEx (Regular Expression) expressions to place signals and modules into dedicated SLR regions that are delimited by these pblocks. An example from this script is shown in Listing 8.1. In this example, BCs (BRCLUSTER instance) are constrained to be mapped to a given SLR region, given a design that adopts a 3-by-3 BCs configuration.

Listing 8.1: An example of some contents of a ¡pre opt_design¿ step placement guidance script [4].

```
add_cells_to_pblock [get_pblocks pblock_dynamic_SLR0]
[get_cells -hierarchical -filter NAME=~*/dragon_top_inst/accelerator_inst/gen_rows[*]
    .gen_cols[0].BRCLUSTER*]
add_cells_to_pblock [get_pblocks pblock_dynamic_SLR1]
[get_cells -hierarchical -filter NAME=~*/dragon_top_inst/accelerator_inst/gen_rows[*]
    .gen_cols[1].BRCLUSTER*]
add_cells_to_pblock [get_pblocks pblock_dynamic_SLR2]
[get_cells -hierarchical -filter NAME=~*/dragon_top_inst/accelerator_inst/gen_rows[*]
    .gen_cols[2].BRCLUSTER*]
```

8.3.3 Reducing SLLs for lower power dissipation and enhanced scalability

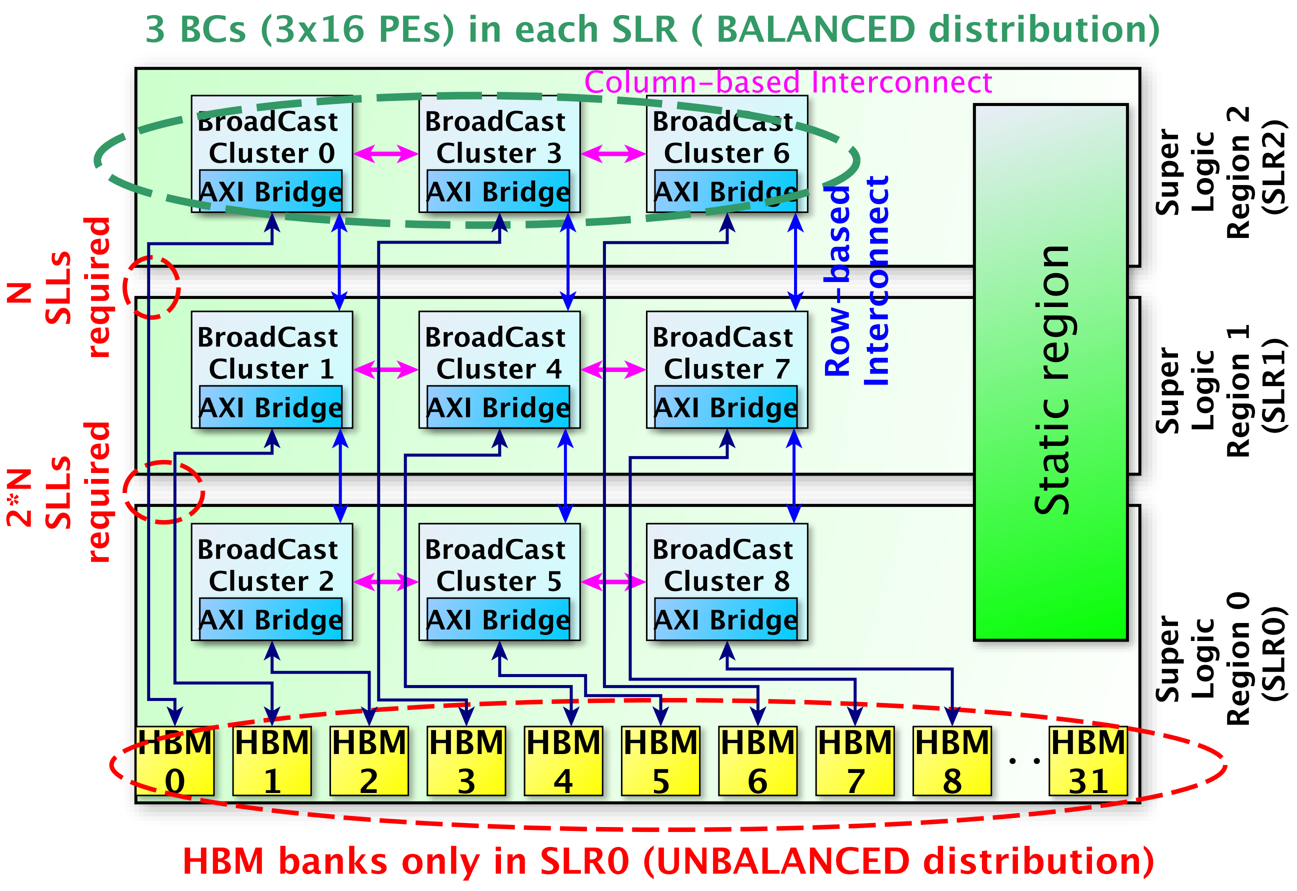


Figure 8.6: Interconnect limitation in SLR boundaries due to the unbalanced requirement on the number of SLLs.

HBM-enabled multi-die FPGAs, such as the one embedded in the Xilinx Alveo U280 data-center acceleration card [37], consists of three dies that are called SLR (Super Logic Region) which are packaged in a single chip and connected to each other using special wires called SLLs (Super Long Lines) [111]. While the the three SLRs contain a comparable amount of hardware resources, the

HBM memory banks reside uniquely in the bottom region SLR0.

To achieve an optimal resource utilization, an overlay must span the exact amount of PEs across the three SLR regions. Optimally, the number of HBM banks in all three regions would be equal. Nevertheless, the unbalanced distribution of HBM memory banks creates a bottleneck regarding the amount of SLL wires required to connect them to each cluster of PEs. In fact, the amount of SLLs required to connect HBM banks to clusters residing in SLR1 and SLR2 is double the amount of SLLs required to connect them to clusters residing only in SLR2, which is a limiting factor for the overlay's scalability, as shown in Fig. 8.6.

The overlay contains multiple BCs that are organized in a $\mathrm{N_{COL}}$ columns by $\mathrm{N_{ROW}}$ rows configuration. An equal amount of BCs is placed into each SLR region, where each BC interfaces with a corresponding HBM bank through an AXI4 bridge.

The available amount of SLLs in the target FPGA is equal to 23,040 and can be computed using Eq.8.1, where $\mathrm{N_{LagColSLL}}$ is the amount of SLLs in every Laguna column (equal to 1,440), $\mathrm{N_{LagCol}}$ is the amount of Laguna columns in every clock region (equal to 2) and $\mathrm{N_{ClkReg}}$ is the amount of clock regions in every SLR, next to the boundary crossing, (equal to 8).

$$\mathrm{N_{SLL\_available}} = \mathrm{N_{LagColSLL}} \times \mathrm{N_{LagCol}} \times \mathrm{N_{ClkReg}}. \tag{8.1}$$

The overlay's Controller part is placed in the bottom SLR0 and uses DMA engines and an AXI interface to exchange data between BMs and their corresponding HBM memory banks. The vast majority of wires in the AXI interface are used by the data buses in the write and read channels. this means the number of wires required to exchange data is approximately $2 \times W_{\mathrm{AXIBus}}$ where $\mathrm{W_{AXIBus}}$ is the bus width in one direction of transfer.

Furthermore, additional SLL wires are required to exchange data between BCs, nearby SLR boundaries. The exact number of these wires is defined by the topology and degree of the interconnect. For example, in a 2D-Mesh interconnect, every pair of BCs residing on both sides of an SLR boundary crossing, contains 4 PEs facing each other on each side. Each pair of PEs is connected through a 64-bit input and another 64-bit output. In this case, $\mathrm{W_{Row\text{-}based\text{-}InterconnectBus}}$ evaluates to 512 (which is 2 x 64 x 4).

Fig. 8.7 shows how this value will be quadrupled in the case of a 4D-Mesh interconnect. The number of required SLLs ($N_{\text{SLL\_required}}$) is defined by the SLR0-to-SLR1 crossing bottleneck (assuming BCs are evenly placed in every SLR) which leads to Eq.8.2 (where N_{SLR} is the amount of available SLRs and is equal to 3 in the target FPGA).

$$N_{\text{SLL\_required}} \geq N_{\text{ROW}} \times \left[W_{\text{Row-based-InterconnectBus}} + (N_{\text{SLR}} - 1) \times (2 \times W_{\text{AXIBus}} \times \frac{N_{\text{COL}}}{N_{\text{SLR}}}) \right]. \quad (8.2)$$

Eq.8.2 leads to Eq.8.3 that defines the AXI data bus upper bound width, considering the number of dies (N_{SLR}), the cluster configuration (N_{ROW} and N_{COL}) and the interconnect bus width ($W_{\text{Row-based-InterconnectBus}}$).

$$W_{\text{AXIBus}} \leq \left[\frac{1}{2}\right] \times \left[\frac{N_{\text{SLR}}}{N_{\text{SLR}} - 1}\right] \times \left[\frac{N_{\text{SLL\_available}} - (N_{\text{ROW}} \times W_{\text{Row-based-InterconnectBus}})}{N_{\text{ROW}} \times N_{\text{COL}}}\right]. \quad (8.3)$$

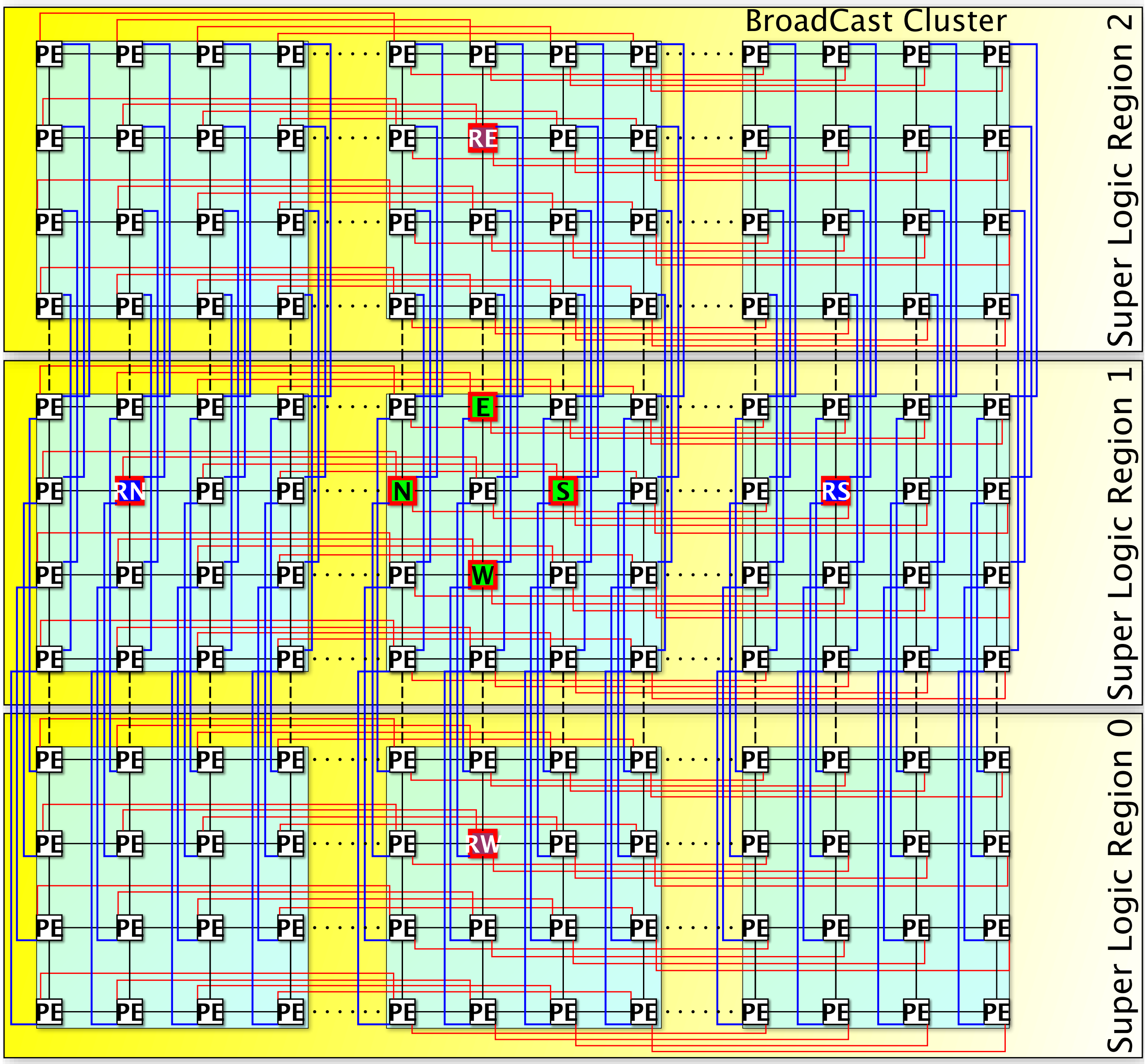
Knowing the available number of SLLs and SLRs in a given HBM-enabled multi-die FPGA target, the analysis of Eq.8.3 allows to deduce the maximal AXI data bus width for a given configuration of BCs. As a result, the 1,024-bit width of the AXI data bus has been decreased to 256-bit width, leading to an optimal overlay scalability as will be shown in the experimental evaluation in Chapter 9.

8.3.4 Layout-aware interconnect generation

The number of neighbors that can directly exchange data with a given PE defines the dimension of the overlay's interconnect. An example showing the connections between PEs in 2D, 3D and 4D-Mesh interconnects is depicted by Fig. 8.7, for an overlay consisting of 9 BCs (9x16 PEs).

- **2D-Mesh interconnect** Here, all BCs (and all PEs) are mapped into a 2D virtual grid structured as 3x3 BCs (12x12 PEs). Every PE can directly exchange data with the surrounding four neighboring PEs in the N (North), S (South), E (East) and W (West) directions, as depicted by Fig. 8.7.

- **3D-Mesh interconnect** Here, an extra 3$^{\text{rd}}$ dimension is added to the interconnect. The overlay is logically re-organized in a 3D grid consisting of 3 2D-grids (a 2D grid is logically structured as 4x12 PEs), where each 2D grid spans three SLR regions. The direct 2D connection between BCs belonging to the same SLR region is now replaced with re-



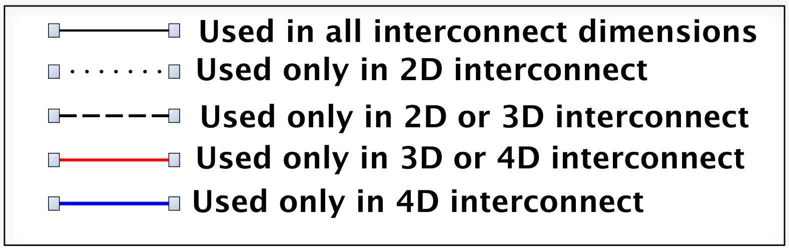

Used in all interconnect dimensions
Used only in 2D interconnect
Used only in 2D or 3D interconnect
Used only in 3D or 4D interconnect
Used only in 4D interconnect

Figure 8.7: Example wiring between processing elements in a 2D or a 3D or a 4D Mesh interconnect [4].

mote connections alongside the RN (Remote North) and RS (Remote South) horizontal directions, as shown in Fig. 8.7.

The horizontal layout of RN and RS neighbors is explicitly chosen to avoid SLR crossing by keeping the remote connections inside the same SLR. This allows to maintain an equal amount of SLL wires as compared to that used in the 2D interconnect. Ultimately, in this 3D configuration, the 9-BC (144-PE) overlay is configured as 1x3x3 BCs (4x12x3 PEs).

- **4D-Mesh interconnect** Here, an extra 4$^{\text{th}}$ dimension is added to the interconnect. This replaces the existing link between each pair of PEs that are facing each others alongside the SLR boundary by remote connections following the vertical direction illustrated in Fig. 8.7, where each PE is now connected to a RE (Remote East) neighbor and RW (Remote West) neighbor. Ultimately, in this 4D configuration the 9-BC (144-PE) overlay is configured as 1x1x3x3 BCs (4x4x3x3 PEs).

8.4 Summary

This chapter summarizes the key micro-architecture enhancements implemented by DRAGON2 and DRAGON2-CB, that led to performance and power-efficiency improvements over the Baseline DRAGON overlay implementation on the target FPGA [37]. Among these enhancements, a novel compact buffering scheme has been introduced that reduces the memory footprint of communication buffers to nearly 50%. Moreover, the DRAGON2 overlay benefits from a manually designed floorplan that is aware of the different regions in the FPGA as well as the scarce number of wires connecting these regions. Furthermore, the operating clock speed has been improved due to the deeper pipeline of the PE and its MAC FPU. The related implementation results will be presented in detail in Chapter 9.

Part IV

Results and Discussion

Chapter 9

Experiments and Results

9.1 Introduction

This chapter presents the evaluation methodology of the DRAGON architecture and its different micro-architecture implementations. The benchmarks used for the experimental evaluation are presented in detail, then, experimental results including computational performance, power-efficiency, EPR, and scalability are provided and discussed, showing the effects of the introduced enhancements on the different micro-architecture implementations.

9.2 Evaluation Benchmarks and Setup

9.2.1 Experimental Setup for the Baseline DRAGON Overlay

The baseline DRAGON overlay is implemented on the Alveo U280 Xilinx data center acceleration card [37] and achieves 130 MHz (144 PEs and 2D-Mesh interconnect).

The Alveo U280 card used in the experiments, contains a multi-die HBM2-enabled FPGA with three SLRs (Super Logic Regions) and 8 GB on-chip HBM2 memories that are split into two 4GB-stacks, with a total of sixteen banks in each stack.

Table. 9.1 provides the setup details of the experimental environment.

Table 9.1: Environment setup for the experimental evaluation of the Baseline DRAGON overlay [3].

CPU (FPGA Host)	Intel Core i9-9900K CPU 3.60GHz (8 cores, TDP=95W)
OS (Operating system) (FPGA Host)	Ubuntu 18.04.1 LTS
Compiler (FPGA Host)	g++(7.5.0) (with -fopenmp -O3 -mavx2)
Accelerator	Alveo U280 Data Center Accelerator Card [37]
FPGA Compiler	Xilinx Vitis 2020.2 (64-bit)
CPU (PC Host)	Intel Core i5-6360U CPU 2.00GHz (2 cores, TDP=15W)
Operating system (PC Host)	MacOS Catalina 10.15.6
Compiler (PC Host)	clang++(10.0.1) (with -fopenmp -O3 -mavx2)

Table 9.2: Environment setup for the experimental evaluation of DRAGON2 and DRAGON-2CB [4].

CPU (Host)	Intel Core i9-9900K CPU 3.60GHz (8 cores) 64GB DDR4 RAM			
OS (host)	Ubuntu 18.04.1 LTS			
FPGA	Alveo U280 Data Center Accelerator Card [37]			
Framework	Xilinx Vitis 2020.2 (64-bit)			
Compiler	Vivado 2020.2 (64-bit)			
Vivado	**Opt_design**	**Place_design**	**Route_design**	**Physical_Opt_design**
Strategies	ExploreWithRemap	ExtraTimingOpt	NoTimingRelaxation	ExploreWithAggressiveHoldFix

9.2.2 Experimental Setup for DRAGON2 and DRAGON2-CB

The experimental evaluation of DRAGON2 and DRAGON2-CB micro-architecture implementations is performed using the same target FPGA [37], that was used also for the evaluation of the Baseline DRAGON overlay. Details of the setup used for the experimental evaluation are given by Table 9.2. Here, Xilinx Vitis was used to compile the various design versions of DRAGON2 and DRAGON2-CB that are using different amount of PEs and interconnect degrees.

9.2.3 Experimental Evaluation

9.2.3.1 Concepts of Iterative Stencil Computing

Iterative stencil computing belongs to a special algorithm category where the data can be organized in a grid that has N-dimensional structure of multiple points. These points depend on their surrounding neighbor points with respect to a particular pattern that is known as a stencil.

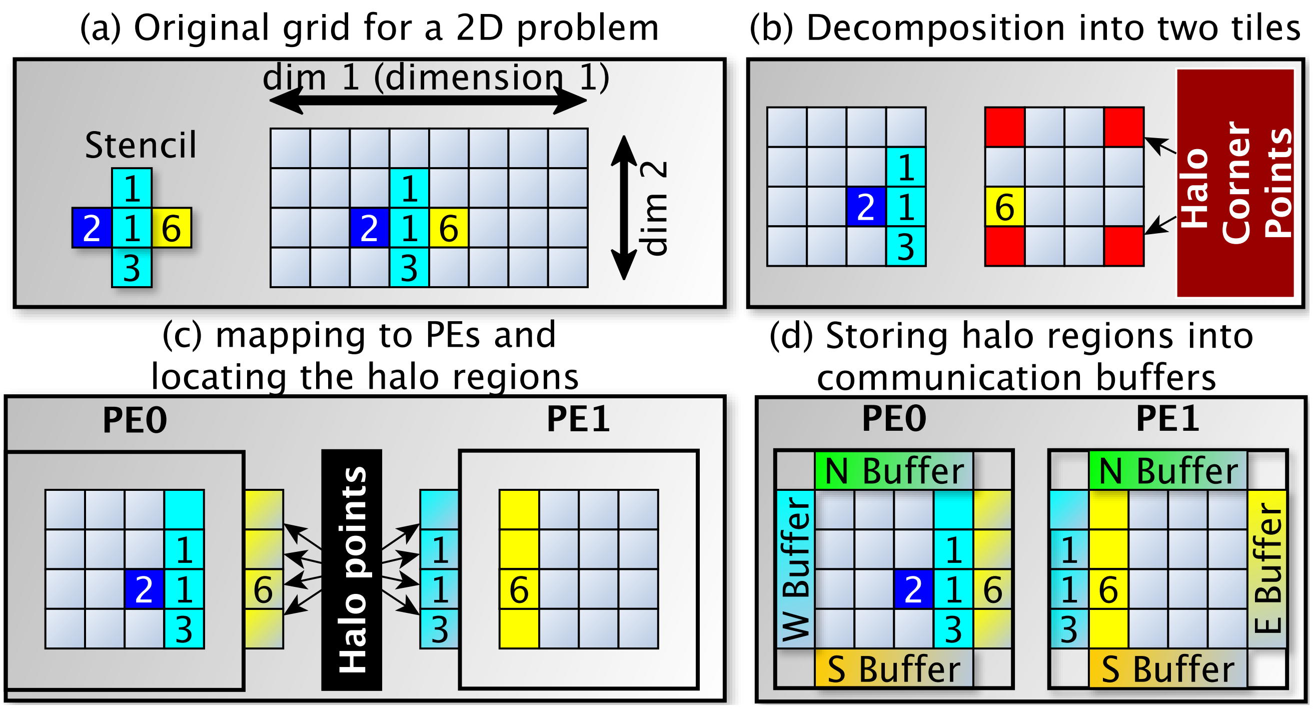

(a) Original grid for a 2D problem
dim 1 (dimension 1)
Stencil
dim 2
(b) Decomposition into two tiles
Halo Corner Points
(c) mapping to PEs and locating the halo regions
PE0
PE1
Halo points
(d) Storing halo regions into communication buffers
N Buffer
W Buffer
S Buffer
E Buffer

Figure 9.1: Concepts of grid partitioning, tiles mapping and halo points exchange through communication buffers in stencil computations [4].

Applications based on the successive iterative update of stencils are commonly used in scientific computations such as the numerical simulation of physical phenomena. Example applications include weather forecasting, computational fluid dynamics [118], tsunami simulation [119]. Other more mainstream applications include as well the processing of image data [120].

The basic computation consists of successively iterating over each point in the original grid by updating their values based on coefficients that are multiplied with the neighboring points in their vicinity that belong to the corresponding stencil pattern.

Laplace and Jacobi equations are model examples of such computations. The 2D version of these examples have a star-shaped stencil, where a central point depends on its surrounding points, in the North, East, South and West directions as depicted by Fig. 9.1.(a). The iterative update model of such equations for N-dimensional grid, where N=2,3 or 4, can be seen in Table 9.3.

In fact, these equations will be implemented as assembly-based programs that are compiled for different grid sizes, in order to be used later for the experimental evaluation of the proposed many-core overlay architecture.

Each of these equations has been implemented in a C program that uses embedded DRAGON ISA assembly opcodes. An example 2D grid of points is illustrated by Fig. 9.1.(a). The original grid points for each benchmark, has been logically split into multiple tiles as depicted by Fig. 9.1.(b) and physically mapped to a corresponding PE (stored in its local memory) as shown in Fig. 9.1.(c). Here, the boundaries of each tile (also called halo points [121]) need to be exchanged with neighboring PEs during the stencil point updates due to the dependence of these points to their neighbors residing in a separate adjacent PE. This can be ensured through the use of neighboring communication buffers (as depicted by Fig. 9.1.(d)), which allows the direct data transfer between PEs, without the need of sending dependent data across all levels of the underlying memory hierarchy.

While Fig. 9.1.(a) shows an example of a 2D grid of stencil points, other stencil problems may be structured in a 3D grid or even have a higher dimension. Nevertheless, the principles of grid decomposition and mapping to multiple PEs, as well as the halo data exchange between PEs through communication buffers remains the same in the implemented benchmarks whose equations are shown in Table 9.3.

9.2.3.2 Evaluation Methodology

The experimental evaluation aims to study the outcomes of the Baseline DRAGON overlay implementation and to analyze the effects of the subsequent improvements that were introduced in the DRAGON2 and DRAGON2-CB micro-architecture implementations. The evaluation also investigates the overlay computational efficiency (EPR), computational performance, power-efficiency as well as scalability. The evaluation approach consists of three steps that are addressed in an incremental order. The first evaluation step aims to perform a comparative study between the baseline DRAGON micro-architecture and the enhanced DRAGON2 micro-architecture (Both micro-architectures adopt the exact regular buffering model), to study and quantify the impact of the introduced enhancements.

Subsequently, the costs as well as the benefits of the introduced compact buffering model (in the DRAGON2-CB micro-architecture) are studied in a comparative manner with the regular buffering scheme (in the DRAGON2 micro-architecture), which constitutes the second step of the evaluation methodology.

The study on both two first steps evaluates the same size overlay (144 PEs) interconnected through a 2D-Mesh topology, from the viewpoint of performance, power efficiency and resource utilization.

For a fair comparison, the same overlay configuration is used during the first two evaluation steps. That is all the micro-architecture implementations of the Baseline DRAGON, DRAGON2 and DRAGON2-CB are evaluated using an overlay implementation with a 2D-Mesh interconnect and a total of 9 Broadcast Clusters (9x16 PEs).

Finally, the last step consists of an in-depth scalability study that includes a comparison of the computational performance, hardware resource utilization, power efficiency, and code size, between the enhanced DRAGON2 micro-architecture and the further-enhanced DRAGON2-CB micro-architecture that adopts an improved buffering scheme.

For the experimental evaluation, the stencil computing benchmarks presented in Table 9.3 have been translated into machines codes that can be executed on each version of the implemented overlays.

Table 9.3: Equation models of the software benchmarks used in the experimental evaluation [3, 4], [9, 10].

Benchmark	**Equation**	#Mul-OPs	#Mul-Acc-OPs	#OPS$_\mathrm{B}$
2D Laplace	$U^{t+1}_{i,j} = (1/4) \times (U^t_{i-1,j} + U^t_{i+1,j} + U^t_{i,j-1} + U^t_{i,j+1})$	1	3	7
2D Jacobi	$U^{t+1}_{i,j} = c_{WEST}.U^t_{i-1,j} + c_{EAST}.U^t_{i+1,j}$ $+c_{CENTER}.U^t_{i,j} + c_{NORTH}.U^t_{i,j-1} + c_{SOUTH}.U^t_{i,j+1}$	1	4	9
3D Laplace	$U^{t+1}_{i,j,k} = (1/6)$ $\times(U^t_{i-1,j,k} + U^t_{i+1,j,k} + U^t_{i,j-1,k} + U^t_{i,j+1,k} + U^t_{i,j,k-1} + U^t_{i,j,k+1})$	1	5	11
3D Jacobi	$U^{t+1}_{i,j,k} = c_{WEST}.U^t_{i-1,j,k} + c_{EAST}.U^t_{i+1,j,k} + c_{CENTER}.U^t_{i,j,k}$ $+c_{NORTH}.U^t_{i,j-1,k} + c_{SOUTH}.U^t_{i,j+1,k}$ $+c_{REMOTE-NORTH}.U^t_{i,j,k-1} + c_{REMOTE-SOUTH}.U^t_{i,j,k+1}$	1	6	13
4D Laplace	$U^{t+1}_{i,j,k,l} = (1/8) \times (U^t_{i-1,j,k,l} + U^t_{i+1,j,k,l} + U^t_{i,j-1,k,l} + U^t_{i,j+1,k,l}$ $+U^t_{i,j,k-1,l} + U^t_{i,j,k+1,l} + U^t_{i,j,k,l-1} + U^t_{i,j,k,l+1})$	1	7	15
4D Jacobi	$U^{t+1}_{i,j,k,l} = c_{WEST}.U^t_{i-1,j,k,l} + c_{EAST}.U^t_{i+1,j,k,l} + c_{CENTER}.U^t_{i,j,k,l}$ $+c_{NORTH}.U^t_{i,j-1,k,l} + c_{SOUTH}.U^t_{i,j+1,k,l}$ $+c_{REMOTE-NORTH}.U^t_{i,j,k-1,l} + c_{REMOTE-SOUTH}.U^t_{i,j,k+1,l}$ $+c_{REMOTE-WEST}.U^t_{i,j,k,l-1} + c_{REMOTE-EAST}.U^t_{i,j,k,l+1}$	1	8	17

9.2.3.3 Evaluation Metrics

To obtain the power dissipation as well as the wall-clock-time relative to the execution of each benchmark, dynamic profiling has been enabled through Vitis prior to the bitstream generation of each overlay (FPGA compilation). Then, during runtime, Vitis generates reports containing these information which can be examined through the Vitis Analyzer tool.

Later, the collected power dissipation and wall-clock-time results for each benchmark on each overlay hardware configuration, can be used to obtain the related sustained computational performance using Eq. 9.3, the related power efficiency using Eq. 9.4 and the related EPR using Eq.9.5.

The TPP (Theoretical Peak Performance) may be computed using Eq.9.1. The terms in Eq. 9.1 hint that increasing the computational performance can be achieved either by increasing the operating clock frequency (FREQ), the amount of Broadcast Clusters ($\mathrm{N_{BC}}$) and/or the amount of PEs in each BC ($\mathrm{N_{PE/BC}}$). The coefficient '2' in Eq. 9.1 comes from the fact that each PE can implement two operations that are a floating-point multiplication and a floating-point accumulation (addition), at every clock cycle.

$$\mathrm{TPP_{DRAGON}\ [FLOPS]} = 2 \times \mathrm{N_{PE/BC}} \times \mathrm{N_{BC}} \times \mathrm{FREQ\ [Hz]} \tag{9.1}$$

The amount of operations required to update each single stencil point $\mathrm{\#OPS_B}$ [FLOPs] (for each stencil benchmark with a given dimension) is provided in Table 9.3. It consists of the sum of the number of multiply-accumulate operations (#Mul-Acc-OPS [FLOPs]) multiplied by 2 (since each multiply and accumulate operation count as a multiplication and a subsequent addition) and the number of multiplications (#Mul-OPs [FLOPs]).

The total amount of operations required by any benchmark to complete the update of all stencil points is computed by Eq.9.2 where $\mathrm{N_{iters}}$ is the amount of all required update iterations for each stencil point, $\mathrm{Size_{Tile}}$ is the partition size of the smaller tiles that form together the original stencil grid and which are stored in each PE's local memory, and $\mathrm{N_{PE}}$ is the total amount of PEs in the implemented overlay.

$$\mathrm{\#OPS_{DRAGON}\ [FLOPs]} = \mathrm{N_{iters}} \times \mathrm{\#OPS_B\ [FLOPs]} \times \mathrm{Size_{Tile}} \times \mathrm{N_{PE}} \tag{9.2}$$

Following the equation Eq. 3.4, the Sustained Performance of DRAGON is given by Eq. 9.3 where $\mathrm{T_{wall\_clock\_time}}$ is the time to complete the update of all the stencil points from the original input grid. This includes the time to read non-processed data and the time to write it back when it is fully processed. This wall-clock-time is collected from runtime profiling report and checked through the Vitis Analyzer software tool.

$$\text{SP}_{\text{DRAGON}}\ [\text{FLOPS}] = \frac{\#\text{OPS}_{\text{DRAGON}}\ [\text{FLOPs}]}{\text{T}_{\text{wall\_clock\_time}}\ [\text{s}]} \tag{9.3}$$

Following the equation Eq. 3.5, the power efficiency of DRAGON is based on Eq. 9.4 and is computed by dividing the Sustained Performance (obtained from Eq. 9.3) by its average power dissipation POWER [W], that is collected, for each benchmark, from its corresponding runtime power profiling report and checked through the Vitis Analyzer software tool.

$$\text{Power-Efficiency}_{\text{DRAGON}}\ [\text{FLOPS/W}] = \frac{\text{SP}_{\text{DRAGON}}\ [\text{FLOPS}]}{\text{POWER}\ [\text{W}]} \tag{9.4}$$

Ultimately, the EPR of DRAGON is based on Eq. 3.6 and computed as the ratio of the Sustained Performance of DRAGON $\text{SP}_{\text{DRAGON}}$ [FLOPS] to its Theoretical Peak Performance $\text{TPP}_{\text{DRAGON}}$ [FLOPS] as shown in Eq. 9.5.

$$\text{EPR}_{\text{DRAGON}}\ [\%] = \frac{\text{SP}_{\text{DRAGON}}\ [\text{FLOPS}]}{\text{TPP}_{\text{DRAGON}}\ [\text{FLOPS}]} \tag{9.5}$$

Note that the equations presented in this section target the DRAGON overlay in general which indeed apply to the Baseline DRAGON overlay, the DRAGON2 as well as the DRAGON2-CB overlays.

9.3 A comparative experimental study of the Baseline DRAGON and the DRAGON2 Overlays

The DRAGON2 micro-architecture aims to improve the shortcomings of the Baseline DRAGON micro-architecture. Here, the results of both implementations are presented in a comparative manner, with regards to area (resource utilization), Sustained Performance and Power Efficiency. This comparison aims to fairly highlight the effects of the applied enhancements; therefore, the implemented DRAGON2 overlay kept the same number of PEs (144 PEs) and the same interconnect degree and topology (2D-Mesh) that were used in the Baseline DRAGON that was proposed in [3].

9.3.1 Effects of the introduced enhancements on resource utilization

Table 9.4: Comparison of the resource utilization between the proposed DRAGON2 and the Baseline DRAGON from [3]. Both overlays are implemented in a 3-by-3 BCs configuration (144 PEs), using a 2D-Mesh interconnect and the same regular buffering scheme [4].

	PE		**BC (16 PEs)**		**OVERLAY (9 BCs)**		**(OVERLAY+FPGA shell)**	
DRAGON:	**2**	Baseline	**2**	Baseline	**2**	Baseline	**2**	Baseline
LUT	2231	3297	36631	54134	352161	505202	514406	657622
LUT mem	251	296	4075	4736	36774	58835	53688	86205
REG	3020	1815	55449	29574	546802	281765	737745	531152
BRAM	6	5	96	80	864	720	1070	940
URAM	1	1	32	32	304	304	304	304
DSP	13	13	208	208	1872	1872	1876	1876

A comparison summary of resource utilization of both the DRAGON2 and the baseline DRAGON overlays is reported by Table 9.4.

While the amount of used URAMs remains identical between both overlays, the number of used BRAMs has slightly increased. In fact, two BRAM memories are used to implement the RegisterFile of DRAGON2 as compared to the baseline version that adopts a mix of a single BRAM coupled with distributed memories. This design choice saves area by reducing other resources such as LUTs and LUTmems, leading to a more compact implementation that aims to decrease the power dissipation.

Besides, the DRAGON2 implements an eight-stage pipelined MAC FPU as compared to the two-stage pipeline of this unit in the baseline version. Consequently, the deeper pipeline allowed the enhanced MAC FPU to use the same amount of DSP resources albeit in a more efficient manner, in particular for the multiplication part, where LUT-based logic used to sum partial products is now appended to the internal multiplication path inside the DSP48E2 instances of the FPGA.

Compared to the Baseline DRAGON, the number of LUTs in the DRAGON2 overlay is dramatically reduced, mainly, as a result of the more compact two-BRAM RegisterFile implementation and the deeper pipeline of the enhanced MAC FPU micro-architecture.

Nonetheless, the amount of utilized registers (REGs) has considerably increased, as a direct result of the PE pipeline that has been stretched from seven stages in the baseline DRAGON to fifteen stages in the DRAGON2 micro-architecture implementation.

9.3.2 Effects of the introduced enhancements on computational performance

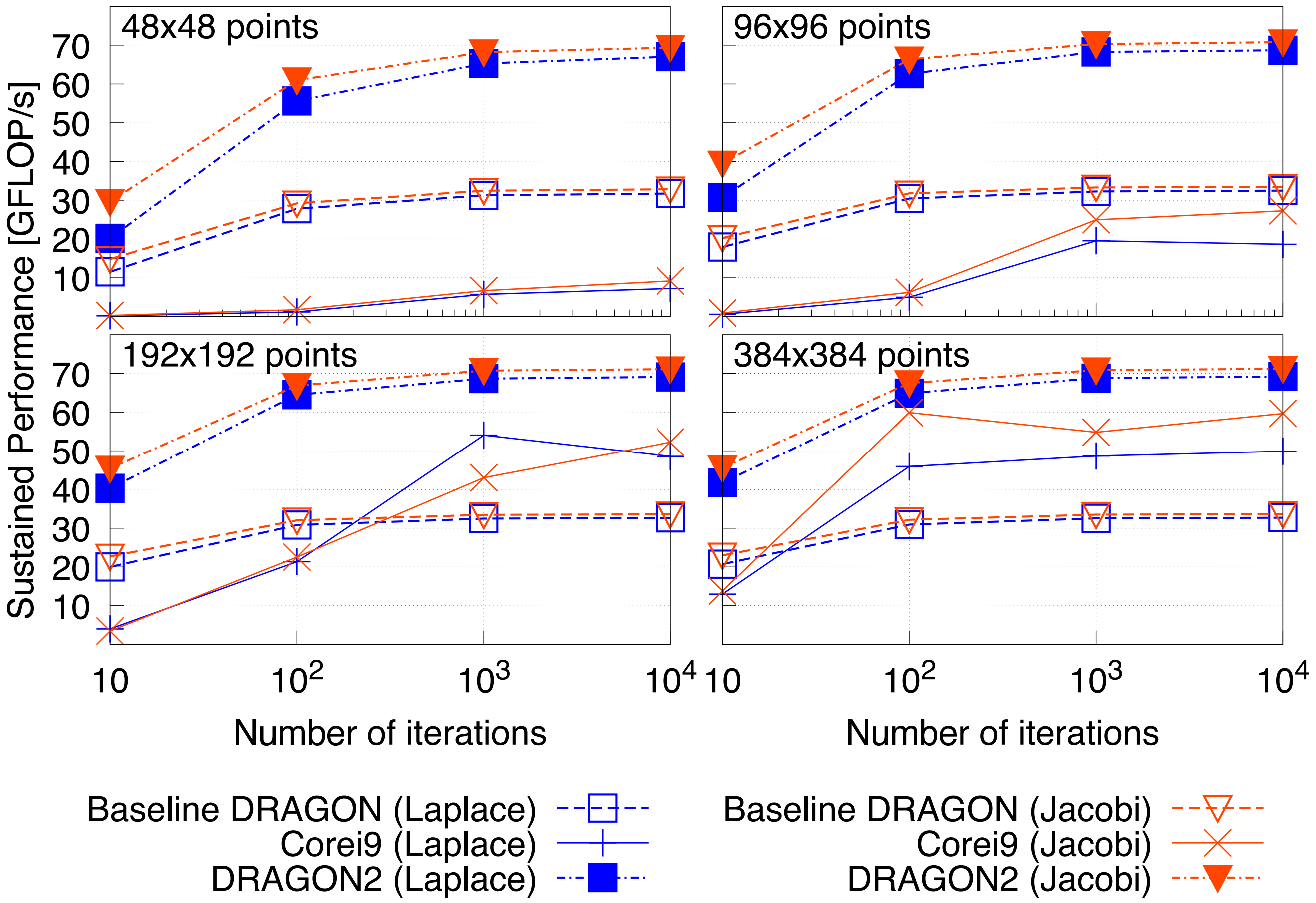


Figure 9.2: Double-precision Sustained Performance of the DRAGON2 overlay as compared to the Baseline DRAGON [4].

The DRAGON2 micro-architecture introduces several enhancements to the Baseline DRAGON architecture, namely, a deeper MAC FPU pipeline (eight stages compared to two in the Baseline DRAGON), a deeper PE pipeline (fifteen stages compared to seven stages in the Baseline DRAGON), a multi-stage pipeline GM-BM AXI data bus (six stages compared to no pipeline stages in the Baseline DRAGON). These enhancements combined allowed the operating clock speed to more than double (from 130 MHz in the Baseline DRAGON to 276 MHz in DRAGON2). Consequently, this drove the DRAGON2 overlay to more than a 100% increase in Sustained Performance as can be seen in Fig. 9.2.

The experimental evaluation of the computational performance in both overlays (DRAGON and DRAGON2) is based on the 2D Laplace and 2D Jacobi benchmarks, whose equations are presented in Table 9.3. These benchmarks are iterative stencils, therefore, the experiments were conducted using different number of iterations (10, 100, 1,000 and 10,000). Moreover, these benchmarks are also two-dimensional, therefore, the original problem grid sizes (48x48, 96x96, 192x192 and 384x384) were split into multiple equally-sized 2D tiles that were stored in each PE's local memory.

A higher number of iterations or a larger 2D tile size leads to better hiding the initial and final overhead cost of moving the respective initial non-processed and final processed data. Thus, it results in achieving near optimal Sustained Performance. Nonetheless, the larger HBM AXI data bus width on the Baseline DRAGON overlay helps it reach its optimal Sustained Performance, faster than the DRAGON2 overlay.

Ultimately, Fig. 9.2 includes as well the results of an Intel Core i9 CPU (9900K) running the same benchmarks[3] and serving as a reference. These results were obtained using software implementations that were optimized for parallel processing, thanks to the use of OpenMP directives and compilation flags that infer the use of AVX2 instructions.

9.3.3 Effects of the introduced enhancements on power efficiency

The double-precision power efficiency of DRAGON2-CB as compared to DRAGON2, for 2D Jacobi and 2D Laplace stencil benchmarks, is depicted by Fig. 9.3, for multiple grid sizes and multiple number of compute iterations.

For a reference with equivalent benchmarks using a CPU-based optimized software implementation, I included the results obtained with an Intel Core i9 9900K, for the same grid sizes and number of update iterations. The CPU-based equivalent programs were designed using OpenMP while enabling the highest optimizations and the use of AVX2 instructions during compilation [3].

The extended pipeline in the enhanced PE micro-architecture has reduced the logic levels in each stage which reduced the probability of glitches and their consequent power dissipation and also allowed targeting higher clock speeds (two times higher than the Baseline DRAGON), which

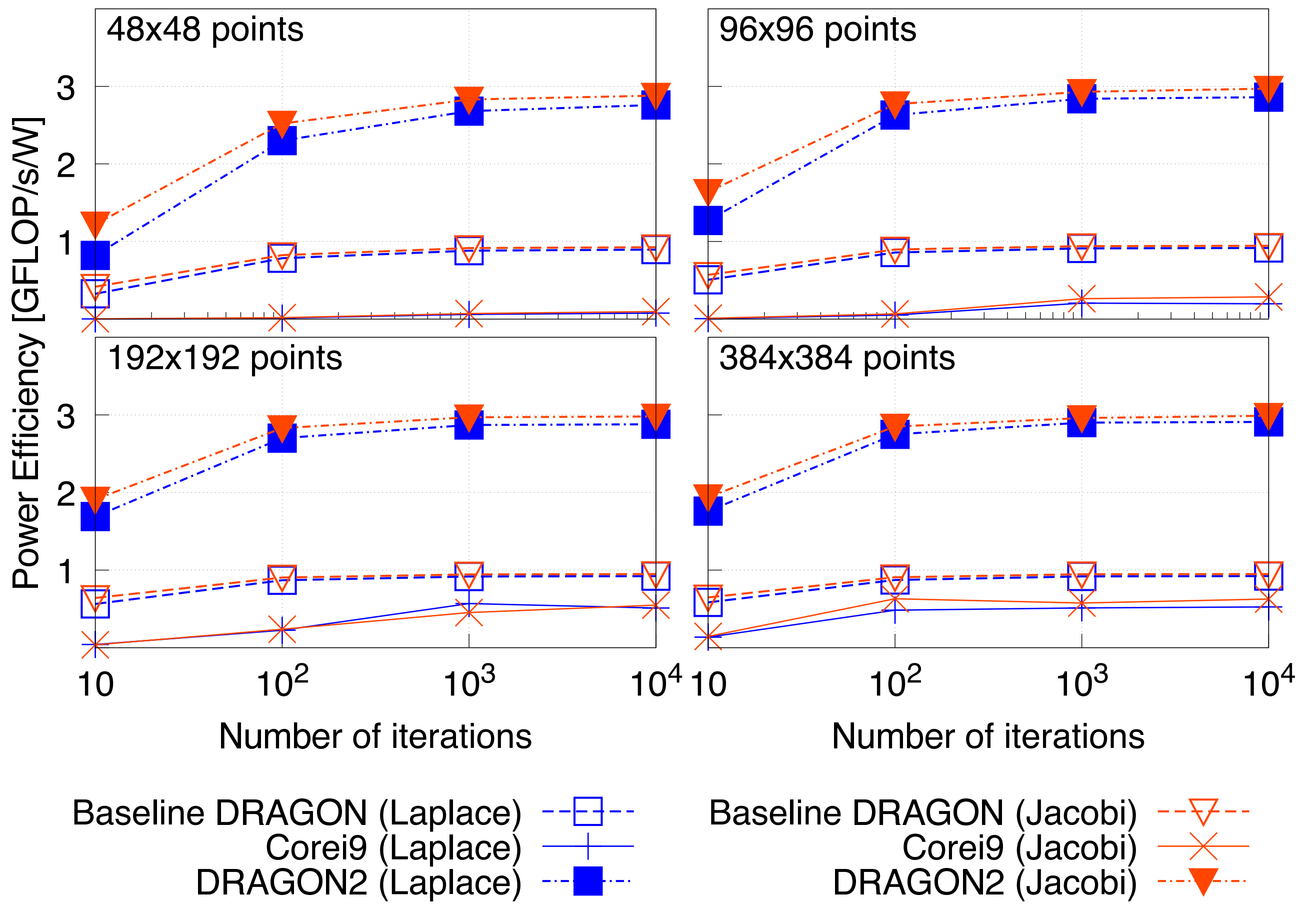


Figure 9.3: Double-precision Power Efficiency of the DRAGON2 overlay as compared to the baseline DRAGON [4].

drove the computational performance to slightly more than two times improvement. Besides, the reduction of the AXI data bus that connects each Global Memory bank (implemented using a HBM bank) to its corresponding Broadcast Cluster, from 1,024 bits to 256 bits, in each data transfer direction (read/write), has significantly reduced the inter-die wires utilization in DRAGON2 as compared to the baseline version.

All these factors have led to approximately three times Power Efficiency improvement as compared to the baseline DRAGON as can be seen in Fig. 9.3

9.4 A comparative experimental study of the DRAGON2 and the DRAGON2-CB Overlays

The DRAGON2-CB micro-architecture aims to enhance further the DRAGON2 micro-architecture, thanks to a compact buffering scheme that aims to improve the power-efficiency by means of lowering the power dissipation as a result of reduced utilization of BRAMs. Here, the results of both implementations are presented in a comparative manner, with regards to area (resource utilization), Sustained Performance and Power Efficiency (energy efficiency). This comparison aims to fairly highlight the effects of the applied enhancements; therefore, the implemented DRAGON2-CB overlay kept the same number of PEs (144 PEs) and the same interconnect degree and topology (2D-Mesh) that were used in the DRAGON2 overlay proposed in [4], as well as the baseline DRAGON overlay that was proposed in [3].

9.4.1 Effects of the compact buffering scheme on resource utilization

The compact buffering scheme dramatically reduces BRAM utilization in the DRAGON2-CB overlay. In fact, the amount of BRAM used for the implementation of four communication buffers with the four neighbouring PEs, can be reduced from four to only one.

As a result, the number of BRAMs per PE is reduced from six to just three, knowing that the RegisterFile maintains the same implementation that consumes two BRAMs. That means the amount of BRAM is halved in all of the PEs as compared to the DRAGON2 overlay implementation, as can be seen in Table 9.5.

The light-weight buffering scheme also reduces the amount of used registers. In fact, the single output of a BRAM in the compact buffering version as compared to the four BRAM outputs in the regular buffering version, leads to a single data operand to be pipelined (registered) instead of four, when passing the Decode stage to the first Execute stage as shown in Fig. 8.3 and Fig. 8.2.

Nonetheless, while the multiplexing logic has been simplified in the first stage of execution, due to the single possible operand coming from the BRAM, the amount of LUTs remains comparable between the DRAGON2 and DRAGON2-CB implementations because of the added logic in the BRAM upstream side, to handle the multiplexing logic for read and write operations related to each internal communication buffer.

Table 9.5: Resource utilization of the DRAGON2-CB overlay which adopts the proposed compact buffering scheme as compared to the DRAGON2 overlay which uses the regular buffering scheme [4].

	PE		**BC (16 PEs)**		**OVERLAY (9 BCs)**		**OVERLAY+FPGA shell**	
DRAGON:	2	**2-CB**	2	**2-CB**	2	**2-CB**	2	**2-CB**
LUT	2231	2235	36631	37824	352161	361361	514406	533626
LUTmem	251	243	4075	3947	36774	35615	53688	52529
REG	3020	2787	55449	51714	546802	514158	737745	732245
BRAM	6	**3**	96	**48**	864	**432**	1070	**638**
URAM	1	1	32	32	304	304	304	304
DSP	13	13	208	208	1872	1872	1876	1876

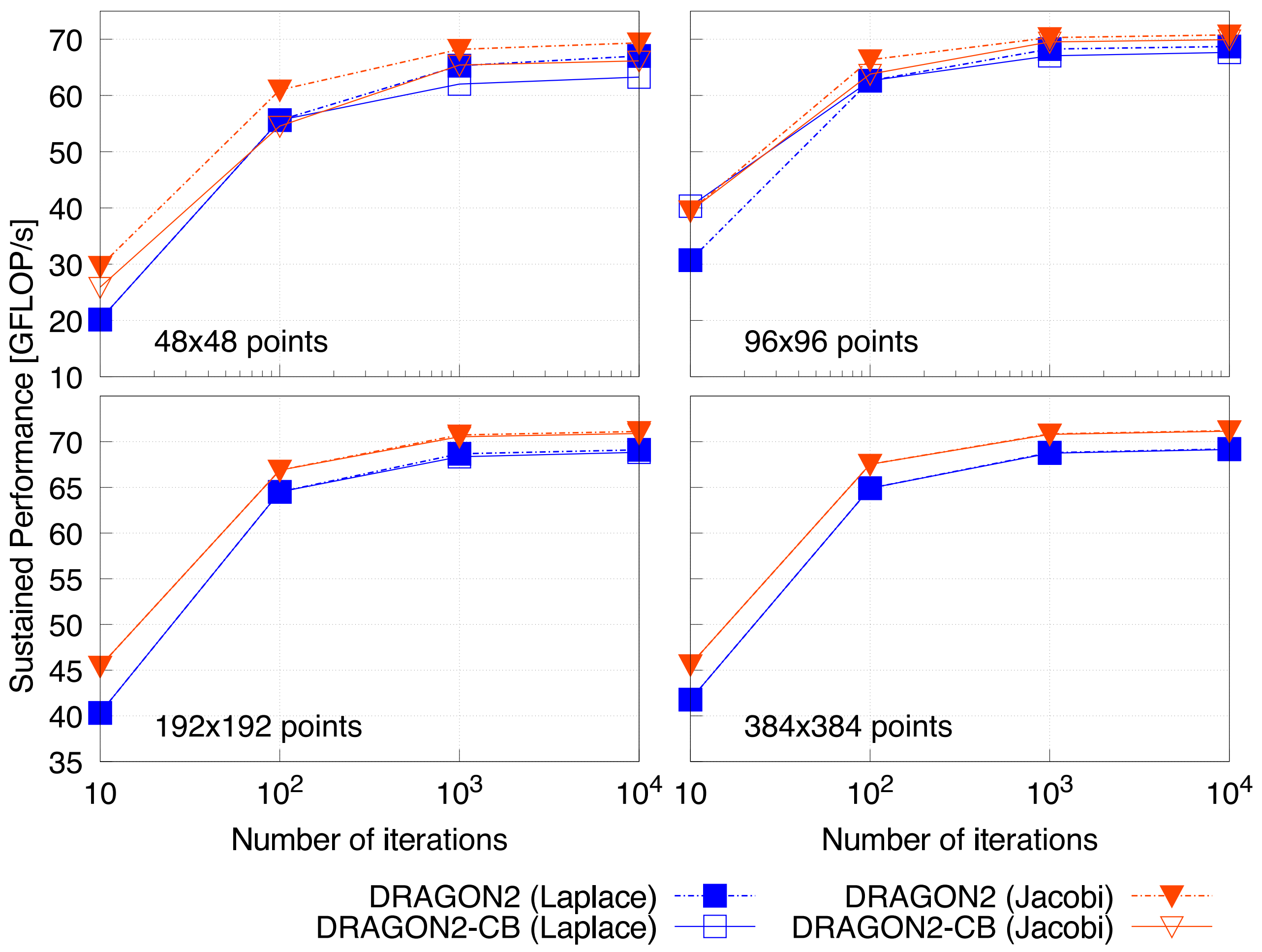


Figure 9.4: Double-precision Sustained Performance of the DRAGON2-CB overlay as compared to DRAGON2 [4].

9.4.2 Effects of the compact buffering scheme on computational performance

A slight drop in the computational performance of the DRAGON2-CB is observed as compared to the DRAGON2 (71.15 GFLOPS as compared to 71.21 GFLOPS with a 2D Jacobi benchmark for a grid size of 384x384), as depicted by Fig. 9.4.

This is in fact due to the use of compact buffering which imposes separate instruction issue of corner data scattering towards neighboring PEs. Since a tile stored in the local memory of a PE has four corner stencil points, four extra clock cycle are required to scatter these data towards North, East, South and West PE neighbors, due to the single write port available in the BRAM holding the four communication buffers of each direction of transfer.

Since only four points need separate clock cycle to issue the additional scattering instructions, the impact of this added time on the performance may be reduced further when increasing the tile size, which reduces the percentage of time lost in the extra scattering operation as compared to the total time to update all of the tile's points, for a given update iteration.

9.4.3 Effects of the compact buffering scheme on power efficiency

The DRAGON2-CB adopts a similar micro-architecture as compared the DRAGON2 overlay with the exception of a light-weight buffering scheme. Hence, it inherits all of the improvements that were introduced in the DRAGON2 as compared to the Baseline DRAGON micro-architecture.

The adopted compact buffering scheme reduces the BRAM utilization of the DRAGON2-CB. In fact, a single BRAM is used to implement communication buffers in the 2D interconnect as compared to four BRAMs in DRAGON2 with the same degree of interconnect. When counting the overall BRAM utilization on the FPGA (including FPGA shell), the amount is reduced from 1070 to just 638 for the 144-PE overlay implementation, which can be seen in Table 9.5).

As a result the power efficiency of the 144-PE DRAGON2-CB is improved by around 12% (with the 2D Jacobi benchmark for a grid size of 384x384) as compared to that of the 144-PE DRAGON2 overlay, with a 2D interconnect, which can be seen in 9.5, where DRAGON2-CB and DRAGON2 achieve at best 3.37 GFLOPS/W and 2.99 GFLOPS/W, respectively.

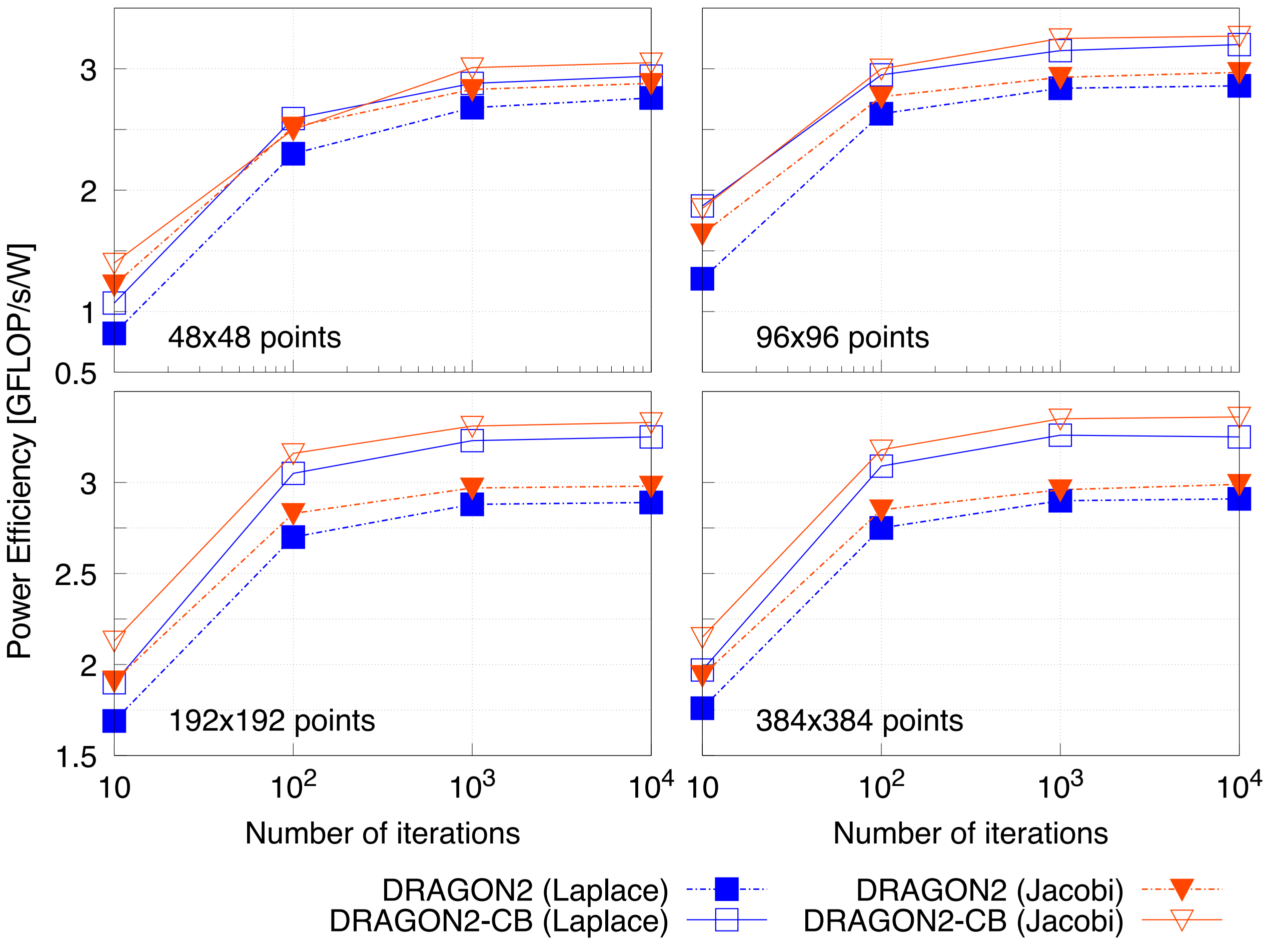


Figure 9.5: Double-precision Power Efficiency of the DRAGON2-CB overlay as compared to DRAGON2 [4].

9.5 A Comparative scalability study of DRAGON2 (Regular Buffering) and DRAGON2-CB (Compact Buffering) Overlays

In this section, an in-depth analysis of the costs and benefits of the compact buffering scheme is given through an extensive comparison study of the DRAGON2-CB overlay that implements such a scheme and the DRAGON2 overlay that implements a regular buffering scheme. This study explores the scalability of each micro-architecture and reports detailed findings, mainly about the area impact (resource utilization), computational performance, power-efficiency (energy-efficiency) and EPR.

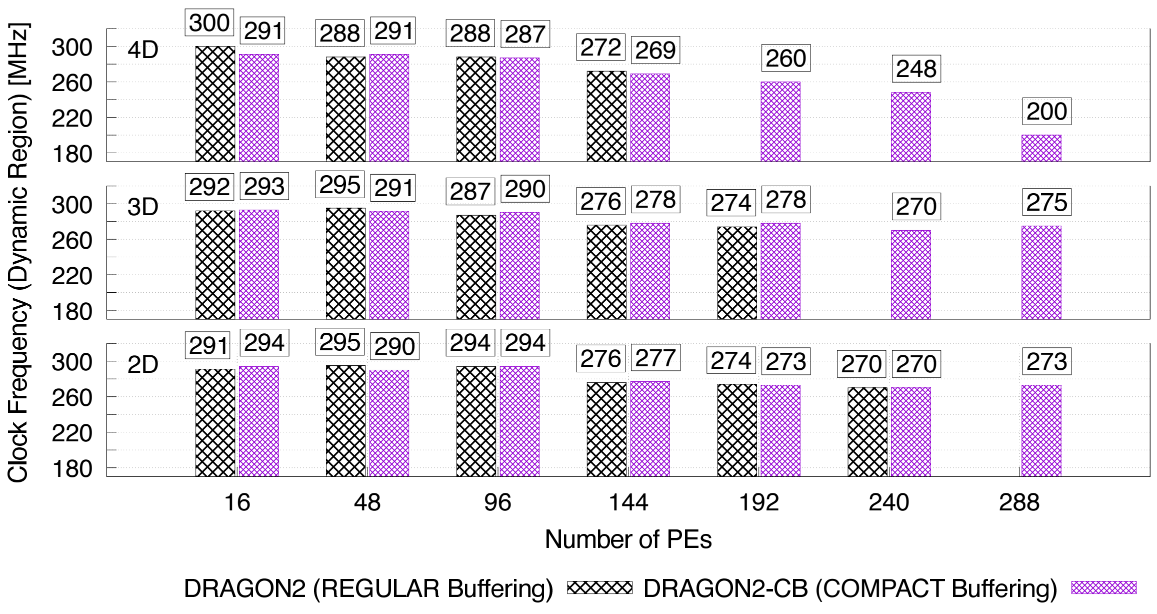


Figure 9.6: Achieved clock speed of DRAGON2 and DRAGON2-CB for 2D, 3D and 4D interconnects with varied overlay size configurations [4].

9.5.1 Impact of the compact buffering scheme on the clock frequency

The compact buffering scheme requires less BRAMs and thus less BRAM outputs. As a result, the large multiplexers of the regular -buffering-PE are simplified as can be seen in Fig. 8.5. Nonetheless, extra control and multiplexing logic is now moved upstream to manage the read/write operations of each communication buffer. Therefore, a scalability study has been conducted to quantify and analyze the impact on the operating clock speed of the overlay.

This study consists of increasing the number of PEs for each implementation using either 2D, 3D or 4D interconnect and reporting the obtained clock frequency after a successful bitstream generation, for both regular and compact buffering versions (DRAGON2 and DRAGON2-CB overlays, respectively).

The reported results are illustrated in Fig. 9.6 which shows a similar trend of operating clock speed on both types of buffering. In fact, while a negligible difference still exists, it is most likely due to the non-determinism of the synthesis and implementation software tool (Vivado).

Interestingly, during run-time the reported design clock speeds has been truncated down to the nearest multiple of five, as noticed through the FPGA runtime profiling reports. For instance, these reports show that a design bitstream capable of operating at 294 MHz will be provided a 290 MHz clock during runtime, while another design that can achieve 300 MHz would maintain its same clock speed.

9.5.2 A study of area (hardware resource utilization) and scalability

Table 9.6: Overlay configurations and related stencil sizes used in the scalability analysis [4].

	2D-Mesh (Tile size=32x32)			**3D-Mesh (Tile size=8x8x12)**			**4D-Mesh (Tile size=4x4x3x10)**		
Number of PEs	**BC config**	**PE config**	**Total stencil size**	**BC config**	**PE config**	**Total stencil size**	**BC config**	**PE config**	**Total stencil size**
16	1x1	4x4	128x128	1x1x1	4x4x1	32x32x12	1x1x1x1	4x4x1x1	16x16x3x10
48	1x3	4x12	128x384	1x3x1	4x12x1	32x96x12	1x1x1x3	4x4x1x3	16x16x3x30
96	2x3	8x12	256x384	1x3x2	4x12x2	32x96x24	1x1x2x3	4x4x2x3	16x16x6x30
144	3x3	12x12	384x384	1x3x3	4x12x3	32x96x36	1x1x3x3	4x4x3x3	16x16x9x30
192	4x3	16x12	512x384	1x3x4	4x12x4	32x96x48	1x1x4x3	4x4x4x3	16x16x12x30
240	5x3	20x12	640x384	1x3x5	4x12x5	32x96x60	1x1x5x3	4x4x5x3	16x16x15x30
288	3x6	12x24	384x768	1x6x3	4x24x3	32x192x36	1x1x3x6	4x4x3x6	16x16x9x60

Ideally, the utilization of any kind of hardware resource on an FPGA-based design should maintain equal percentages for optimal scalability. In other words, an over-utilization ratio for a specific kind of hardware resources would result in a poor scalability that will be delimited by this most consumed resource. The BRAM resources were in fact the bottleneck for scalability, in the Baseline DRAGON as well as the DRAGON2 overlays. In these overlays, a higher interconnect degree coupled with an increased number of PEs, has led to consuming BRAMs significantly faster as compared to other kind of hardware resources. This is indeed visible through a faster slope depicting the percentage of BRAMs used to implement the regular buffering in Fig. 9.7. This bottleneck is even more significant in the overlays having a higher degree of interconnect (3D and 4D versions), where a larger number of BRAMs is required. In some situations, the lack of sufficient resources, needed to meet this requirement, prohibited the deployment of a higher number of PEs.

The DRAGON2-CB benefits from the compact buffering model that effectively reduces the number of BRAMs used to implement communication buffers in 2D, 3D or 4D interconnects. This led to enhancing the DRAGON2-CB overlay scalability and spared sufficient BRAMs resources to be used along with a higher number of PEs and/or a higher degree of interconnect, as shown in Fig. 9.7 and Fig. 9.8 which suggest a better utilization balance and a linear increase trend in

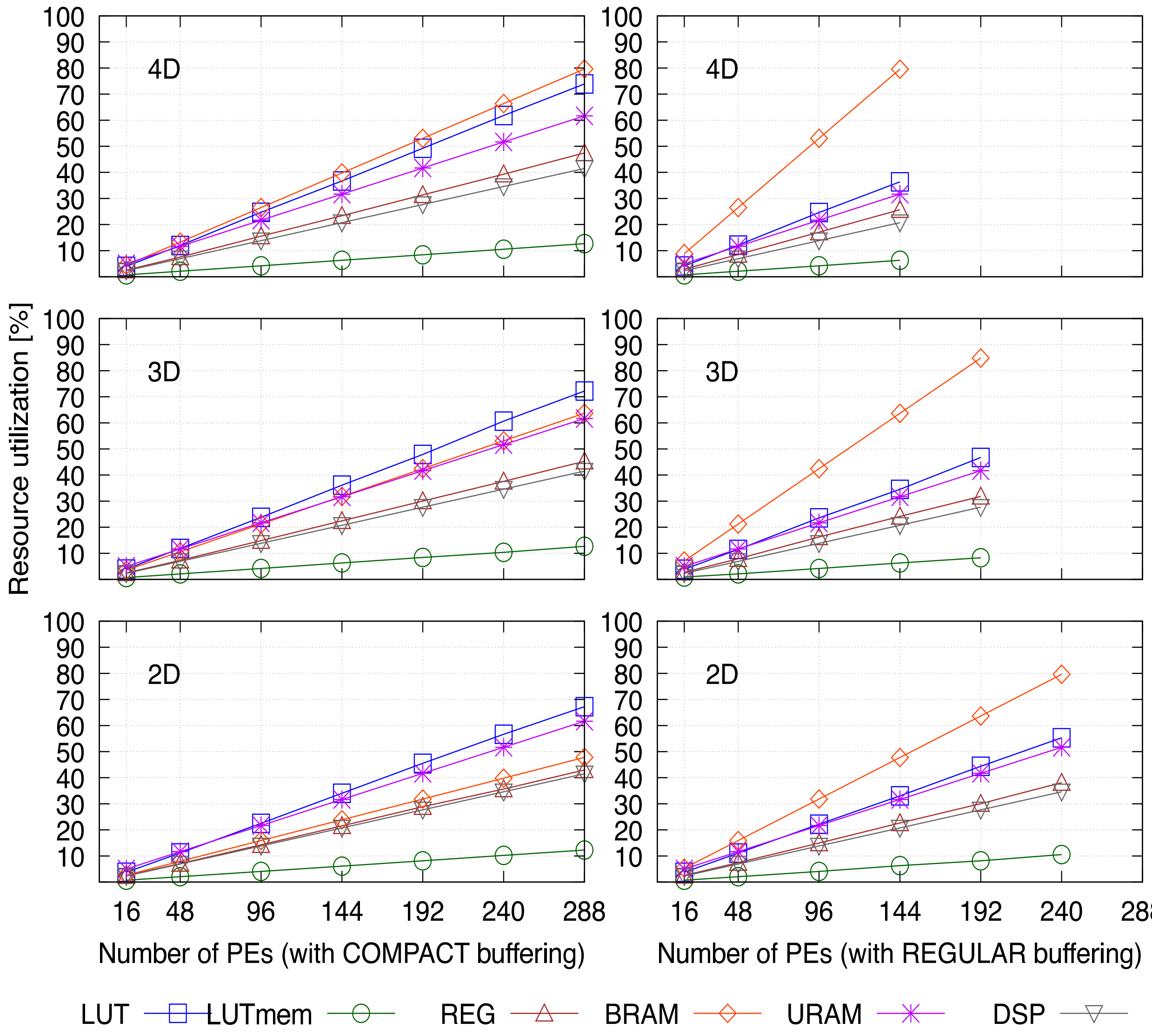


Figure 9.7: Percentage of resource utilization of the proposed DRAGON2-CB (with COMPACT buffering) as compared to DRAGON2 (with REGULAR buffering), for multiple overlay size configurations and with varied dimensions (2D, 3D and 4D) of the Mesh interconnect [4].

terms of the absolute utilized amount of the depicted kinds of hardware resources.

9.5.3 A study of performance, EPR and scalability

The Sustained Performance results are reported in Fig. 9.9 and Fig. 9.10 and are computed based on Eq. 9.3 that involves the wall clock time results obtained from the benchmarks execution profiling reports that were generated by Vitis during runtime. The different configurations for the Broadcast Clusters and their Processing Elements, for each interconnect degree are summarized in Table 9.6. Moreover, this table reports the amount of stencil points of each tile, as well as their total amount in each given dimension, based on the total number of tiles that is equal to

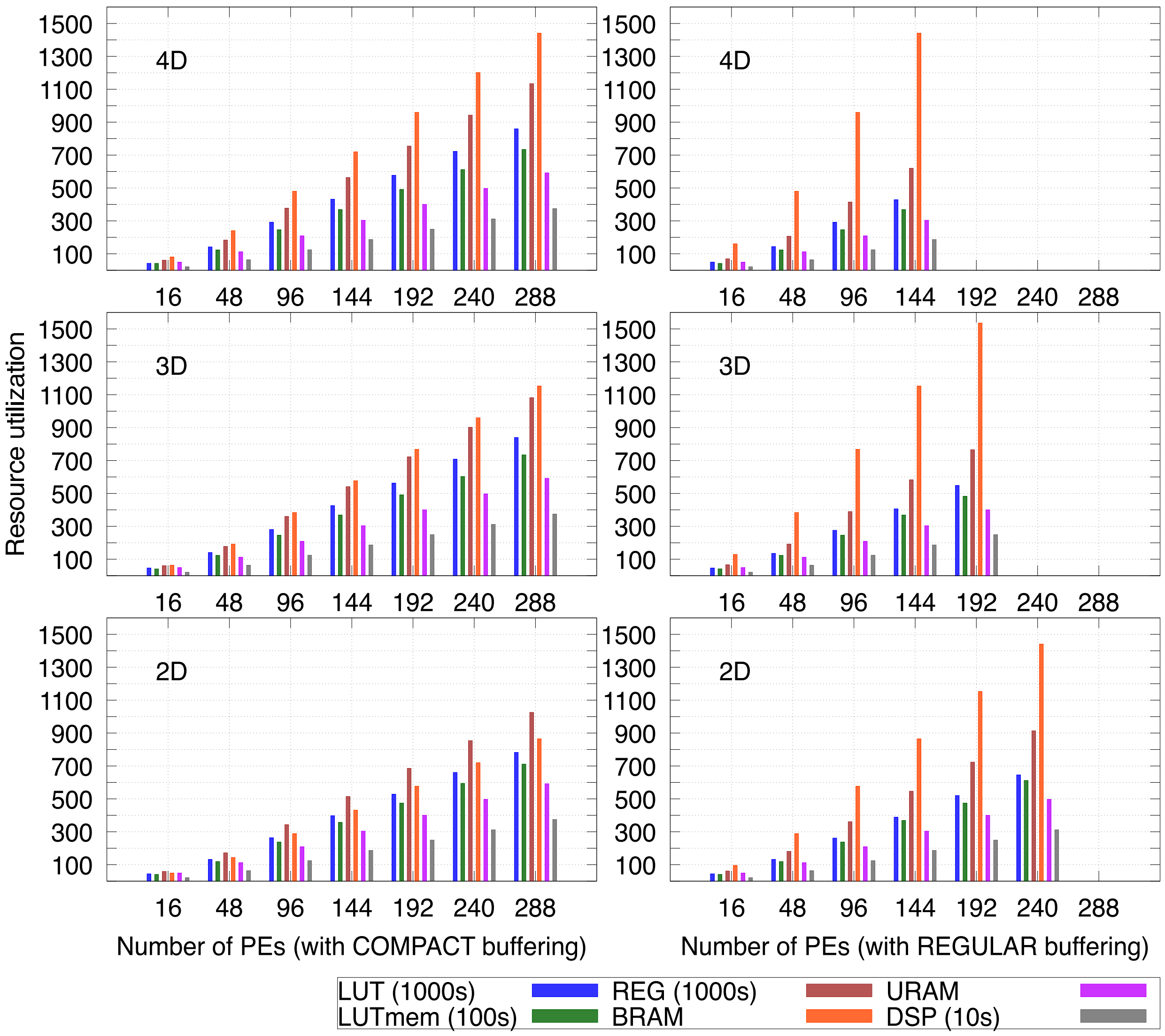


Figure 9.8: Resource utilization of the proposed DRAGON2-CB (with COMPACT buffering) as compared to DRAGON2 (with REGULAR buffering), for multiple overlay size configurations and with varied dimensions (2D, 3D and 4D) of the Mesh interconnect [4].

the total amount of PEs in each configuration. The total number of stencil update iterations in each benchmark is set to 10,000. To study the scalability of the proposed architecture, the total amount of stencil points is increased alongside the increase in the amount of Processing Elements. A nearly linear speedup can be observed in Fig. 9.9 for N-D Jacobi stencil benchmarks and in Fig. 9.10 and for N-D Laplace stencil benchmarks, where N=1,2 or 3. Nonetheless, a performance saturation can be seen for both 4D benchmarks, in the 288-PE DRAGON2-CB overlay with a 4D interconnect. This saturation is caused by the degradation of the operating clock frequency (The operating clock achieves 200 MHz, after about 30 hours of implementation time, from which 25 hours were consumed during the routing step alone). In fact, the growing

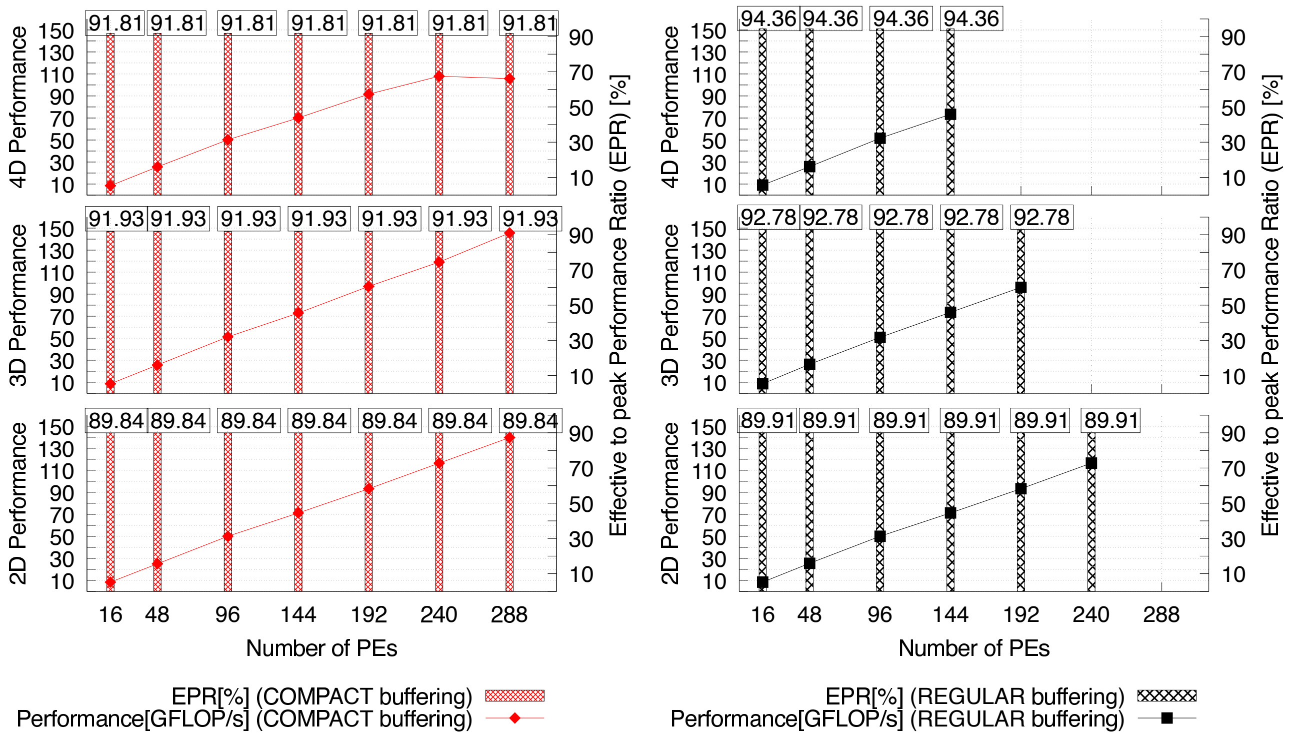


Figure 9.9: Double-precision floating-point performance scalability and Effective to peak Performance Ratio, using 2D, 3D and 4D Jacobi benchmarks. A side-by-side comparison between DRAGON-2CB (COMPACT buffering) and DRAGON2 (REGULAR buffering) [4].

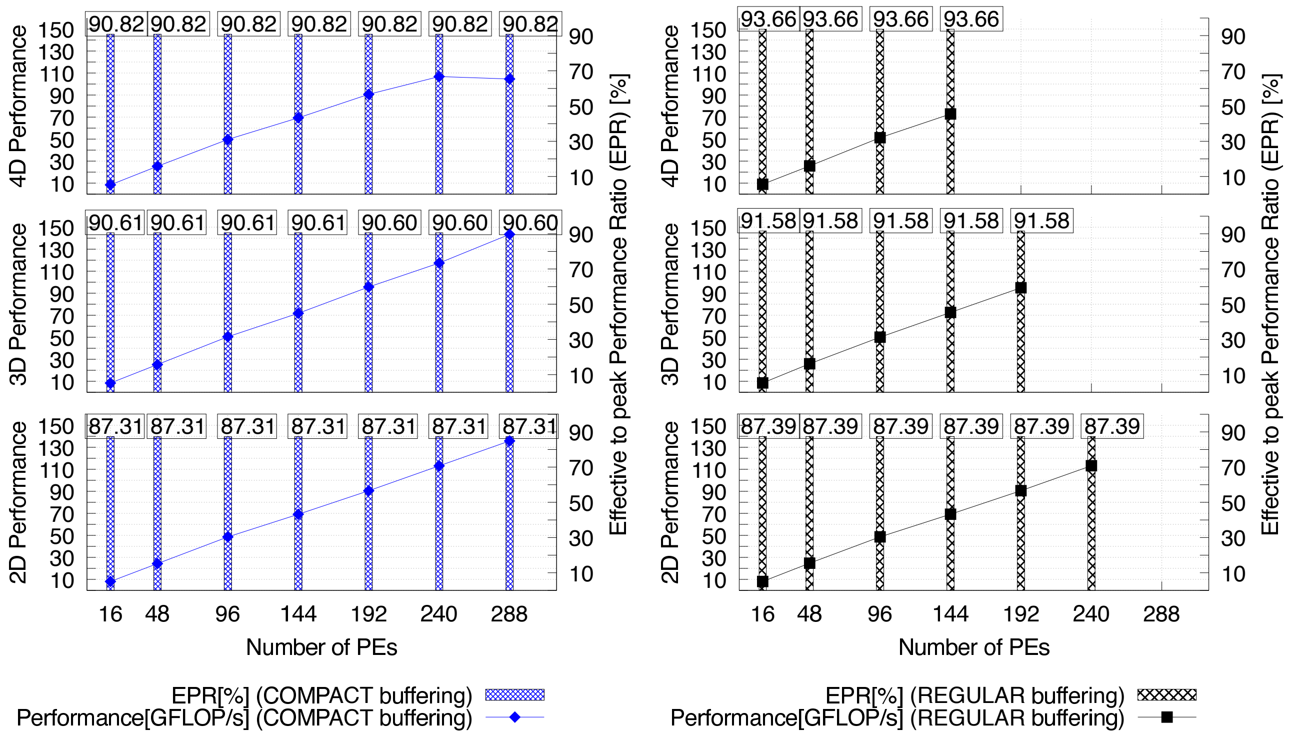


Figure 9.10: Double-precision floating-point performance scalability and Effective to peak Performance Ratio, using 2D, 3D and 4D Laplace benchmarks. A side-by-side comparison between DRAGON-2CB (COMPACT buffering) and DRAGON2 (REGULAR buffering) [4].

number of PEs coupled with a growing degree of interconnect, increases the amount of wires that cross the SLR boundary regions and complicates the routing due to the scarcity of the SLL wires connecting the multiple die regions on the FPGA.

Besides, the DRAGON2-CB implementation aims to improve the power efficiency (energy efficiency) by reducing the power dissipation through a reduction of the amount of required BRAMs, while maintaining comparable Sustained Performance results with the DRAGON2 overlay (that uses a regular buffering scheme) by maintaining a comparable EPR.

The EPR results are reported in Fig. 9.9 and Fig. 9.10) and are computed based on Eq. 9.5 that derives from Eq. 9.3 and Eq. 9.1. Fig. 9.9 and Fig. 9.10) show that the EPR maintains nearly the same value when increasing the amount of PEs for a given interconnect dimension which confirms a good performance scalability. The range of variation of the obtained EPR remains near constant within a two decimal digits of precision. The high EPR is a reflection of the high computational efficiency of the architecture where a small percentage of the theoretical peak performance is lost in operations that are not involved in the computation such as data movements.

Nonetheless, the extra clock cycles required by the DRAGON2-CB compact buffering scheme to scatter the corner points of a tile of stencil points in each PE, slightly reduce the EPR as compared to the regular buffering used by the DRAGON2 overlay. However, this slight reduction is still acceptable knowing the obtained advantages in terms of power efficiency that will be shown in the next subsection.

9.5.4 A study of power efficiency and scalability

The power efficiency results are reported in Fig. 9.11 and are computed based on Eq. 9.4 that involves the power dissipation results obtained from the power profiling reports generated by Vitis during runtime.

The physical layout aware floorplanning and guided placement allowed the DRAGON2 and DRAGON2-CB with a 3D interconnect to avoid crossing SLR boundaries to connect PEs in the added third dimension. This has led to a SLL wire utilization that is similar to the same overlays that use a 2D interconnect as can be seen in Fig. 8.7. As a result, the obtained

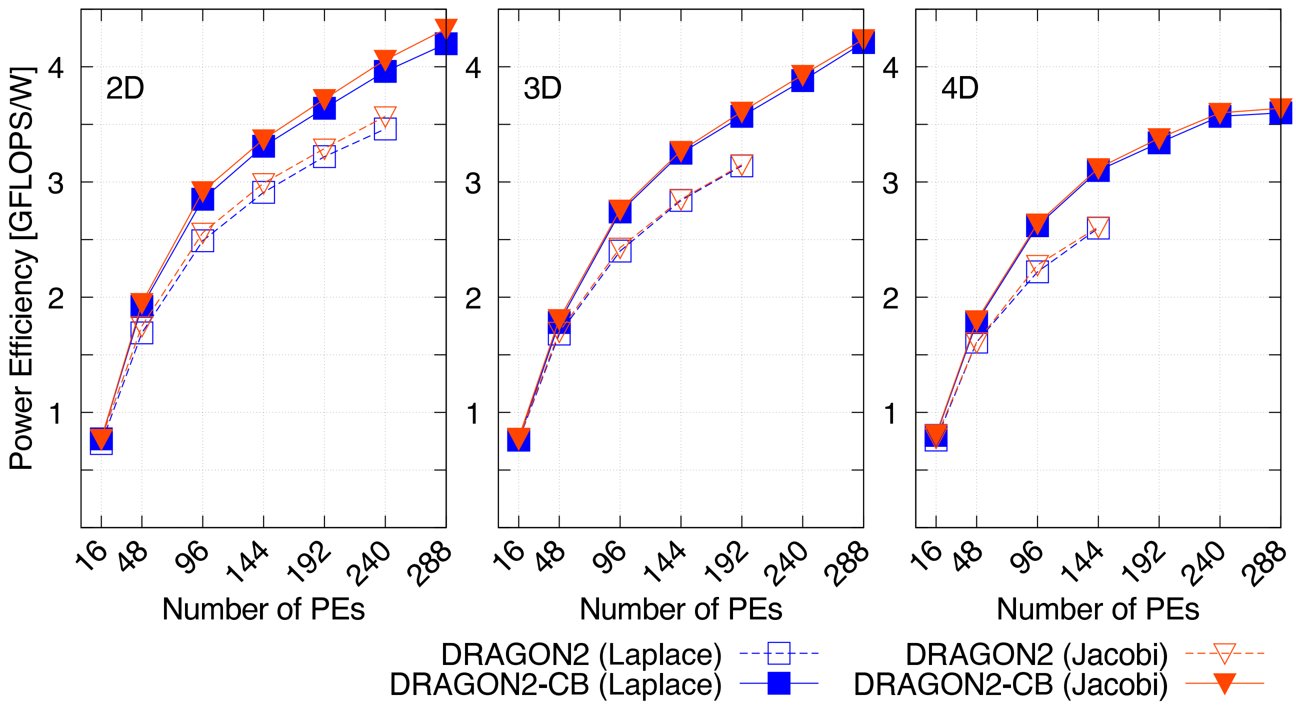


Figure 9.11: Effect of the compact buffering scheme on the obtained Power Efficiency with 2D, 3D and 4D Laplace and Jacobi stencil benchmarks [4].

power efficiency results are comparable despite the 2D version having a slightly better outcome, due to the localized short inter-PE connections as compared to the long connections with the remote-neighbor PEs, in the 3D version.

While the added remote connections for the 3rd dimension in the 3D interconnect can be kept inside the same SLR region, the additional remote connections for the 4th dimension in the 4D interconnect have to cross the boundaries between SLR regions and consequently heavily use the available SLL wires.

In addition to increasing the power dissipation, the excessive use of the inter-region wires, in particular for large-size overlays with more than 192 PEs, complicates routing which limits the operating clock speed and harms the computational performance. The combined effects translate to a degraded power efficiency (energy efficiency). Consequently, the power efficiency of the 4D DRAGON2-CB appears to saturate at slightly above 3.5 GFLOPS/W at sizes beyond 192 PEs, while the 2D and 3D versions of the same sizes seem able to further grow above the 4.2 GFLOPS/W mark.

Thanks to the compact buffering scheme, the DRAGON2-CB has a reduced overall BRAM utilization that leads to a reduced power dissipation and offers not only a better scalability when compared to the DRAGON2, but also an improved power efficiency (energy efficiency). This improvement becomes more visible in Fig. 9.11 when the amount of implemented PEs or the underlying interconnect degree are increased.

9.5.5 Bandwidth and scalability

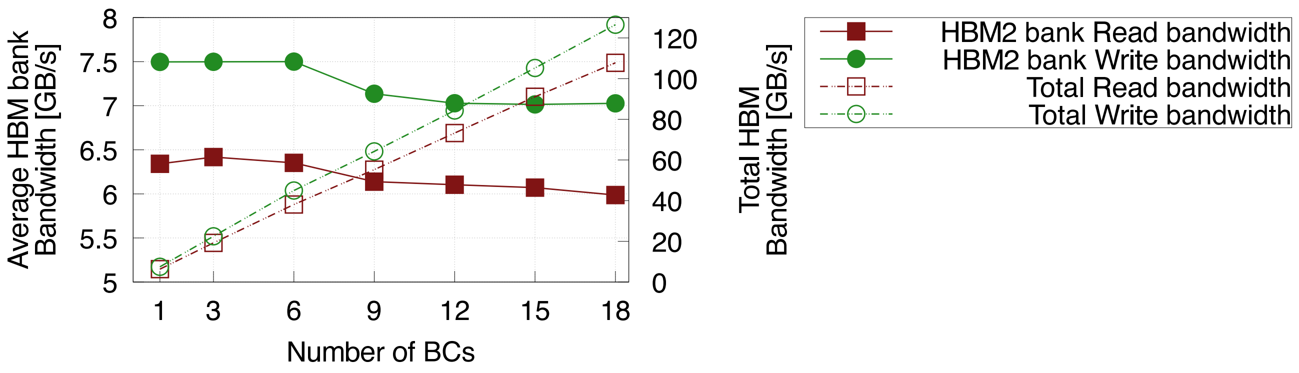


Figure 9.12: HBM2 bandwidth for Read and Write operations [4].

The memory bandwidth of write and read operations for HBM memory banks are obtained by enabling data profiling during implementation and extracting the results from the generated reports. These results correspond to the DRAGON2-CB implementation with a 2D interconnect and a total of 18 BCs. Nonetheless, only the number of BCs impacts these results since the degree of interconnect or the used buffering scheme are only internal to the Accelerator part and do not affect the BM-GM AXI-based data transfer. The memory bandwidth results are illustrated by Fig. 9.12.

Each BC is connected to its corresponding HBM memory bank in a peer-to-peer manner, which allows memory bandwidth scalability alongside an increase in the size of the overlay (increase of the number of BCs). Here, the AXI-based data bus has a 256-bit width and can transfer 32 Bytes or four 64-bit data each single beat, with a maximum of 128 beats per transaction which corresponds to the limits of 4,096 Bytes related to the AXI protocol specification.

Nevertheless, the achieved HBM memory bandwidth findings shown in Fig. 9.12 are in-par with the equivalent burst size results that can be found in previous works such as [122].

9.5.6 Impact on executable code size

The DRAGON2-CB adopts a compact buffering scheme that reduces the number of BRAMs but also the related bandwidth, since all the communication buffers used to transfer data with adjacent neighbors in the 2D space share the same BRAM write port. As a result, issuing a data scattering operation into two different directions among North, South, East or West, will take two different clock cycles instead of one.

An example of these data are the corner points of a stencil tile shown in Fig. 9.13. To handle this situation, two VLIW instructions are required to be issued, one for each direction of data transfer. The additional VLIW instruction is used because an extra clock cycle is needed by the BRAM to write another 64-bit input data coming from a different neighbor. The added VLIW instruction is also coded in 16 Bytes (128 bits or the sum of two 64-bit slots). Consequently, the four corner points in the 2D tile shown in Fig. 9.13 require 4x16 Bytes (64 Bytes) of instruction memory space which corresponds to the difference shown in Table 9.7 for 2D-interconnect overlay mapping the 2D stencil tiles into each LM.

A stencil problem with a higher dimension is mapped to an overlay with the same degree of interconnect to obtain the highest possible performance and power efficiency. Fig. 9.13 shows that a 3D tile consists in fact of multiple 2D tiles stacked together. Consequently the four corner stencil points in the 2D tile have now increased by a_3 times which is the added 3^{rd} dimension of the 3D tile shown in Fig. 9.13. This means the total number of corner points that need to be scattered into two directions has been multiplied by a_3. As a result, $4xa_3$ added extra cycles and extra VLIW instructions are required to perform the corner data transfers. Here, the 3D tile has the dimensions (8x8x12) that corresponds to (a_1, a_2 and a_3), respectively. Therefore, the difference in code size between DRAGON2 (Regular Buffering with 3D interconnect) and DRAGON2-CB (Compact Buffering with 3D interconnect) is reported in Table 9.7 and is equal to 768 (which is the multiplication product of 64 by a_3).

In a similar manner, the data of the 4D benchmarks are decomposed into smaller 4D tiles that are stored in each PE's LM. Here, the original 2D four corner points requiring four separate clock cycles to issue four additional VLIW instructions (one for each corner point) are now multiplied by the added 3^{rd} and 4^{th} dimensions which correspond to 3 and 10, respectively, as shown in

Table 9.7.

Ultimately, it is interesting to note that scaling the compute resources (deploying a larger number of BCs) do not affect the code footprint that maintains a constant size, regardless of the size of the overlay. This size can change though when modifying the size of the data tiles inside LMs.

Table 9.7: Cost of the compact buffering on the size (in Bytes) of the generated binary code for 2,3 and 4D Jacobi and Laplace benchmarks [4].

	2D-Mesh (Tile size = **32x32**)			**3D-Mesh** (Tile size = **8x8x12**)			**4D-Mesh** (Tile size = **4x4x3x10**)		
Benchmark	compact	regular	**Difference**	compact	regular	**Difference**	compact	regular	**Difference**
Laplace	89632	89568	**64**	100544	99776	**768**	89312	87392	**1920**
Jacobi	106080	106016	**64**	112960	112192	**768**	97120	95200	**1920**

9.5.7 Modeling benefits and costs in N-dimensional interconnects

A stencil point can have a shape formed with its neighboring points that belong to 1D, 2D, 3D or further high dimensional space [123]. Table 9.8 provides a summary of the benefits as well as the costs related to the application of the compact buffering scheme in the underlying overlay for a given dimension of its interconnect, based on the implementation results with 2D, 3D and 4D interconnects. A generalization model is provided as well by extrapolating these findings to interconnects having a higher N-D dimension, based on careful analysis of the partitioning of N-D grid into N-D tiles and the impact of the exchange of corner points in these tiles with analogy to the 2D and 3D tiles that are shown in Fig. 9.13..

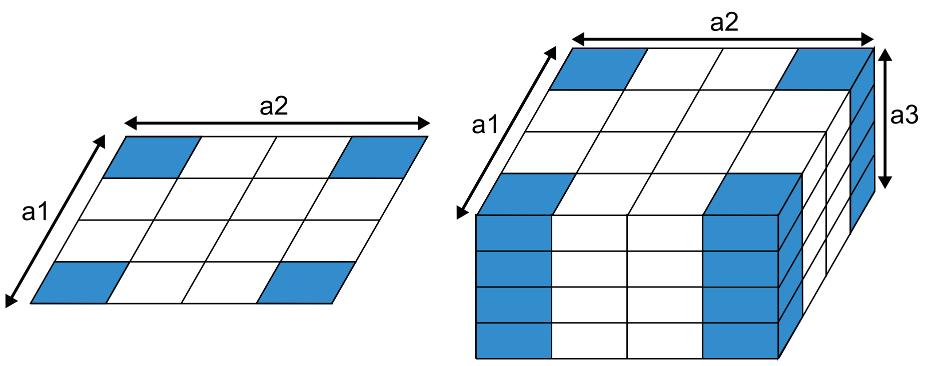


Figure 9.13: 2D (left) and 3D (right) stencil tiles inside a local memory of a PE and their corner points that require extra clock cycles in order to be exchanged with adjacent PEs [4].

After decomposing a 2D stencil problem space into multiple smaller 2D tiles, the boundaries of each of these tiles has to be exchanged with the respective neighbors. In particular, the corner

Table 9.8: Number of required BRAMS in each interconnect dimension and the related clock cycle and code size overhead [4].

Stencil Benchmark and its dimension	**interconnect dimension of DRAGON2**	**#BRAMS (compact) (buffers)**	**#BRAMS (regular) (buffers)**	**required extra cycles (per iteration)**	**program size cost (in Bytes)**
2 D (Laplace/Jacobi)	2 D	1	4	4	64
3 D (Laplace/Jacobi)	3 D	2	6	$4 \times a_3$	$64 \times a_3$
4 D (Laplace/Jacobi)	4 D	3	8	$4 \times a_3 \times a_4$	$64 \times a_3 \times a_4$
N D (Laplace/Jacobi)	N D	N-1	2N	$4 \times \prod_{i=3}^{N} a_i$	$64 \times \prod_{i=3}^{N} a_i$

points of each tile has to be exchanged with exactly two neighbors. Using the compact buffering scheme, a single 64-bit write port is available on the BRAM containing the four communication buffers that exchange these data with the adjacent PEs; Therefore, corner point data has to transferred during two clock cycles instead of simultaneously in the same clock cycle. The 2D tile has $\mathbf{a_1} \times \mathbf{a_2}$ dimensions which are duplicated $\mathbf{a_3}$ times to form a 3D tile for a 3D stencil problem. When using a 3D interconnect, an additional BRAM is used which provides an extra write port and the cost to transfer corner points to neighbors remains the same as in 2D. These points need four additional clock cycles to be scattered towards the neighboring PEs because of the unique 64-bit write port in each BRAM.

Consequently, an increased degree of the interconnect is required to handle an increased problem dimension without degrading its performance. This leads to a formulation of the required extra clock cycles to transfer corner points in an N-D stencil problem, using an N-D interconnect, at each iteration of the computation. This formulation can be seen in Table 9.8 where $\prod_{i=3}^{N} a_i$ is the product of all added dimensions to the original 2D space.

Ultimately, in a 2D problem space mapped to an overlay with a 2D interconnect, the extra VLIW instructions that handle the scattering of corner points require an additional memory space of 4x16 Bytes (64 Bytes). Hence, a N-D problem that is mapped to an overlay with a N-D interconnect, would require $64 \times \prod_{i=3}^{N} a_i$ Bytes, taking into consideration the additional dimensions that duplicate the four corner points of the original 2D space.

9.6 Summary

The DRAGON overlay architecture has various different implementations on the target FPGA [37], namely, the Baseline DRAGON, the DRAGON2 and the DRAGON2-CB overlays. Through the iterative process of building and implementing different micro-architecture features, issues related to the target FPGA started arising, such as the scarcity of wires between the multiple regions in a multi-die FPGA or even the lack of sufficient hardware resources for a specific architectural block. This chapter summarizes, the differences between the various implementations of DRAGON, and provides an in-depth study on each version as well as the impact of each micro-architecture enhancement based on an extensive experimental evaluation.

Chapter 10

Summary and discussion

10.1 Introduction

This chapter summarizes the micro-architectural differences between three implementations of the proposed overlay, namely, the Baseline DRAGON, DRAGON2 and DRAGON2-CB. The summary shows specific micro-architecture implementation choices and reports the best achieved results (for the largest size implementation of these overlays). Besides discussing the experimental outcome of these micro-architecture choices, this chapter also discusses their impact on the programming aspect where the corresponding level of complexity may be increased. Ultimately, a comparison with other studies and their achieved results is presented and thoroughly discussed.

10.2 DRAGON, DRAGON2 and DRAGON2CB overlays, a summary

10.2.1 General Differences

Table 10.1 highlights the main variations among the different implemented versions of the overlay, namely, the baseline version (Baseline DRAGON), the improved version (DRAGON2) and the improved version enhanced further with the compact buffering scheme (DRAGON2-CB).

In fact, the Baseline DRAGON micro-architecture implementation has a limited scalability due to the use of a large 1,024-bit AXI data bus that uses DMA engines to connect the GM banks (mapped to HBM banks) with the BM banks of each BC in a peer-to-peer manner. The subse-

Table 10.1: Comparison between the baseline DRAGON overlay [3], DRAGON2 and DRAGON2-CB overlays [4].

	Baseline DRAGON [3]	**DRAGON2** [4]			**DRAGON2-CB** [4]		
Vivado Implementation	Default	Manually optimized			Manually optimized		
ISA	DRAGON ISA	DRAGON ISA			DRAGON ISA		
ALU pipeline depth	1	2			2		
FPU pipeline depth	2	8			8		
FPU execution	single-thread	multi-thread (4)			multi-thread (4)		
Communication buffers	standard	standard			compact		
PE pipeline depth	7	15			15		
GM-BM AXI bus width	1024-bit	256-bit			256-bit		
Interconnect degree	2D	2D	3D	4D	2D	3D	4D
#PE (largest overlay)	144	240	192	144	288	288	288
Fmax*	130	270	274	272	273	275	200
EPR	89.90	89.91	92.78	94.36	89.84	91.93	91.81
SP [GFLOPS] (Jacobi)	33.66	116.53	96.19	73.37	139.72	145.62	105.77
P.eff [GFLOPS/W] (Jacobi)	0.94	3.57	3.15	2.61	4.33	4.24	3.64

* This is the achieved clock speed of the generated bitstream. The Vitis runtime may slightly reduced it to a multiple of 5. For example, 272 MHz becomes 270 MHz during runtime, while 275 MHz remains the same.

quent reduction of this data bus width from 1,024 bits to 256 bits provided a better scalability which resulted in the possibility to deploy double the number of PEs as compared to the baseline implementation (288 PEs as compared to 144 PEs, respectively).

The deeper pipeline in the MAC FPU of DRAGON2 and DRAGON2-CB imposed a multi-threaded programming approach to extract the best possible compute unit utilization and as such introduced some complexity in programmability. The compact buffering of DRAGON2-CB allows to overcome the scarcity of BRAM resources and allows maintaining the number of PEs in the largest implementation, through 2D, 3D and 4D interconnects. Nonetheless, deploying 288 PEs with a 4D interconnect struggles to maintain the same range of operating clock speed, due to increased routing complexities, and achieves just 200 MHz. While the compact buffering in DRAGON2-CB solves one of the issues facing the scalability of the system, it slightly reduces the achieved EPR as compared to the non-compact buffering version in DRAGON2. Despite the slight decrease in EPR, the DRAGON2-CB remains the superior micro-architecture implementation, thanks to the enhanced scalability that allows it to deploy more PEs on the target FPGA and achieve a better double-precision computational performance and power efficiency.

10.2.2 FPGA-side Programming differences

10.2.2.1 An example program on the Baseline DRAGON overlay

DRAGON adopts attractive architectural aspects to alleviate the cost of data movements on energy consumption and computational performance.

At the system level, it decouples GM accesses from the executed compute instructions, thanks to a custom DAE (Decoupled Access Execute) model.

At the PE level, the architecture allows efficient overlapping of computations with local memory and/or inter-PEs data transfers, thanks to an efficient ISA VLIW encoding that encompasses multiple operations able to be issued in a single clock cycle (For example, two computations such as a multiplication followed by an accumulation and concurrently perform a load from local memory, a store to local memory, as well as scatter and gather data to/from adjacent PEs). This provides the possibility of emulating a cyclic buffer into the RegisterFile that allows the implementation of the sliding window scheme [124] that can be found in applications such as image processing. Fig. 10.1 shows an example that is used in a 2D stencil computing benchmark and that implements this cyclic buffer scheme, through the area marked as "active region". This active region contains the stencil point subject to computation and consists of three lines. The top and bottom lines contain the surrounding North and South points that are used in the calculation, whereas the central line contains the East and West surroundings of the point to be updated. New points data are continuously prefetched from local memory, in a concurrent and cyclic manner along with the update of points from the central line.

A part of the actual program implementation of this cyclic buffer using the DRAGON VLIW instructions is shown through Listing 10.1. This shows a portion from a program that runs on the Baseline DRAGON overlay and performs the mean computation of a stencil point based on its four surrounding North, East, South and West points. In other words, This implements the computations of a 2D Laplace benchmark and makes use of the powerful VLIW concept to efficiently overlap computations with data movements.

10.2.2.2 Compact Buffering impact on the programming of DRAGON2-CB

Listing 10.1 shows a part of the 2D Laplace benchmark that performs the mean computation

Listing 10.1: Using VLIW instructions to implement a sliding stencil cyclic buffer in the Register File of the baseline DRAGON [4].

```
 //-----------------------------------------LEFT-----------
 FMUL(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2L, RDST, OPSRCL, f); NOP(f); //VLIW
      SLOT1 : FMUL instruction. and VLIW SLOT2 : NOP instruction
 //-----------------------------------------RIGHT----------
 FMACCA(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2R, RDST, OPSRCR, f); NOP(f);
 //------------------------------------------DOWN---------
 FMACCA(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2D, RDST, OPSRCD, f); NOP(f);
 //--------------------------------------------UP------
 FMACCA(RSRC1, MODE, LMADDR, BROFFSET, BMADDR, RSRC2U, NDST, OPSRCU, f); LD(LMADDRLD,
      RDSTLD, f);
 //--------------------------------------------------------
 LMADDRLD = (LMADDRLD + 1) % sizeLMdata;
 RDSTLD = (RDSTLD + 1) % sizeRFdata;
 LMADDR = (LMADDR + 1) % sizeLMdata;
 i = i + 1; // increment to jump to the next point in the stencil tile
```

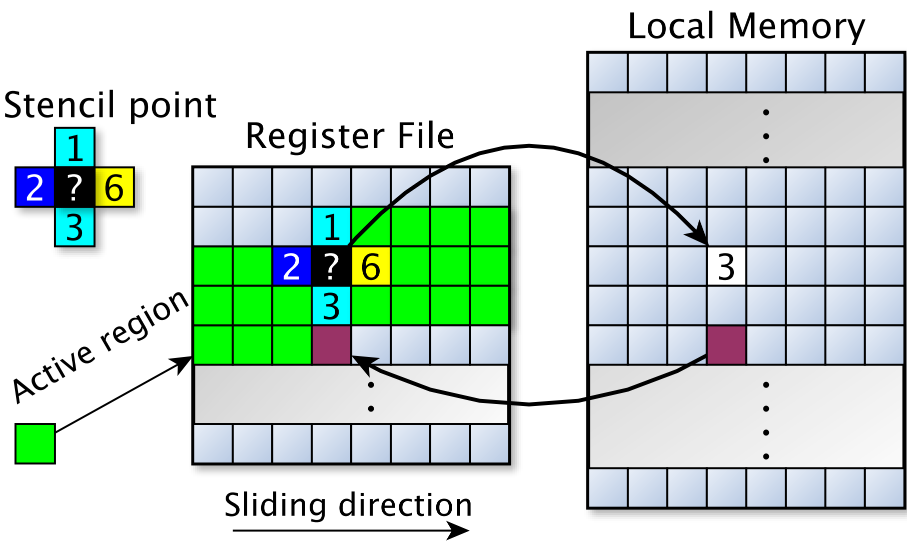


Figure 10.1: Emulation of a sliding window cyclic buffer. Multiple operations may be embedded into a single VLIW instruction (e.g. compute&store to LM + scatter/gather to/from neighboring PEs + load from LM) [4].

for a given stencil point using VLIW instructions. The equivalent program that targets the DRAGON2-CB micro-architecture is shown in Listing 10.2.

To improve the operating speed and reduce glitching, DRAGON2 stretches the MAC FPU pipeline from two stages to eight stages. DRAGON2-CB introduces further enhancement by adopting a compact buffering scheme that reduces the BRAM utilization, thus, leading to reduced area and power dissipation, better energy efficiency and improved scalability.

In contrast, these enhancements come at the cost of an increased difficulty to program the overlay using the same instructions. Simply looking at the size of the two equivalent programs

Listing 10.2: Using VLIW instructions to implement a sliding stencil cyclic buffer in the Register File of the DRAGON2-CB [4].

```
 //-----------------------------------------LEFT-----------
if (j==0){ //First pass on the four-point chunk, Operation : Multiply each point by the same stencil coefficient (C_west)
     FMUL(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2L, RDST, OPSRCL, f); // VLIW SLOT1 ; MODE =0b01 means no scatter towards adjacent PEs
     LD(LMADDRLD, RDSTLD, f); // VLIW SLOT2 ; Load 64bit data from LM @address LMADDRLD to RegisterFile @address RDSTLD
     LMADDRLD = (LMADDRLD + 1) % sizeLMdata; //Local Memory Load address, used for sliding window
     RDSTLD = (RDSTLD + 1) % sizeRFdata; } //Register File target address for LOAD, used for sliding window
//-----------------------------------------RIGHT-----------
if (j==1) //Second pass on the four-point chunk, Operation : Multiply and Accumulate each point by the second stencil coefficient (C_east)
    {FMACCA(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2R, RDST, OPSRCR, f);
     NOP(f);}
//-----------------------------------------DOWN---------
if (j==2) //Third pass on the four-point chunk, Operation : Multiply and Accumulate each point by the third stencil coefficient (C_south)
    {FMACCA(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2D, RDST, OPSRCD, f);
     NOP(f);}
//--------------------------------------------UP------
if (j==3){ //Fourth pass on the four-point chunk, Operation : Multiply and Accumulate each point by the fourth stencil coefficient (C_north)
    if ((NDST==NDST_UP)||(NDST==NDST_UP_and_RIGHT)||(NDST==NDST_UP_and_LEFT)){
      FMACCA(RSRC1, MODE, LMADDR, BROFFSET, BMADDR, RSRC2U, NDST_UP, OPSRCU, f); //First update and scatter towards North (Up) direction only
      NOP(f);
    } else if ((NDST==NDST_DOWN)||(NDST==NDST_DOWN_and_RIGHT)||(NDST==NDST_DOWN_and_LEFT)){//First update and scatter to South (Down) direction only
      FMACCA(RSRC1, MODE, LMADDR, BROFFSET, BMADDR, RSRC2U, NDST_DOWN, OPSRCU, f);
      NOP(f);
    } else { //update and do not scatter data if the current point is not on boundary
      FMACCA(RSRC1, MODE, LMADDR, BROFFSET, BMADDR, RSRC2U, NDST, OPSRCU, f);
      NOP(f);
    }
    if (((i+1)%data_line==0) && ((NDST==NDST_UP_and_RIGHT) || (NDST==NDST_DOWN_and_RIGHT))){ //for example for a 32x32 tile the 'data_line' is equal to 32 //if the current point is a corner point then add an extra data scatter towards West (Left) and East (Right) neighbors.
        NOP(f);
        NSG( 0b10, LMADDR−data_line+1, RSRC2, NDST_LEFT, NSRC, f); //scatter the left corner point to LEFT PE direction
        NOP(f);
        NSG( 0b10, LMADDR, RSRC2, NDST_RIGHT, NSRC, f); //scatter the right corner point to RIGHT PE direction
    }
    LMADDR = (LMADDR + 1) % sizeLMdata; //Local Memory address to store updated points
}//---------------Managing loop indexes inside the tile and the chunk of four-point set
if ((i+1)%4==0) { // Check if the current pass covered all points in the four-point chunk
    if (j!=3) { //Operations on the four-point chunk not completed
       j=j+1 ; // move operation index to next direction (j is the operation index which defines the next coefficient (C_east, C_west, C_north or C_south))
       i=i−3; } // before multiply_and_accumulate with a new coefficient, reset the point index to the start of the four-point chunk
    else { j=0 ; i=i+1; } } //reset operation index to multiply operation (FMUL) and jump to a new four-point chunk.
else //still inside the same four-point chunk
    { i=i+1; } // move point index to next point (i is the point index)
```

in Listing10.1 and Listing 10.2 hints to the added complexity.

In fact, the introduced compact buffering in DRAGON2-CB adopts a single BRAM with a unique write port to store data incoming from the four neighboring PEs in the 2D interconnect. As such, corner points of the 2D tiles (shown in Fig. 9.1.(b)) that were previously scattered in the same clock cycle, now require two clock cycles and an extra VLIW instruction must be issued to perform the additional scattering of data (line 27 to 32 of Listing 10.2).

10.2.2.3 Multi-threading impact on the programming of both DRAGON2 and DRAGON2-CB

DRAGON2 and DRAGON2-CB micro-architectures stretched the MAC FPU pipeline to eight stages. In particular, they stretched the accumulation pipeline to four stages which imposes the use of multi-threading for a full use of the compute capacity. That means populating the pipeline with a chunk of four points at a time instead of performing point by point computations.

Listing 10.2 shows this effect where the (i) index points to the global position of the stencil tile's point under update, whereas the (j) index indicates the ongoing pass atop the chunk of four points under update.

A 2D Laplace stencil point update is equivalent to the mean computation of the four surrounding points. In the Baseline DRAGON, this is ensured through a single multiplication of one surrounding point by the value 0.25 (1/4) that is followed subsequently by three multiply-and-accumulate operations.

In contrast, the DRAGON2 and DRAGON2-CB micro-architectures have to fully populate the accumulation pipeline stages in order to extract the highest possible performance; therefore, four (j) passes are required over all the surrounding North, South, East and West directions (j = 0,1,2 and 3). At first (j=0), four multiplications with 0.25 have to be performed for each point in the four contiguous stencil points. Then, at each iteration (j=1 or 2 or 3), four multiply-and-accumulate operations follow in a consecutive manner to perform the stencil point update for the current active direction.

10.2.2.4 The DRAGON2-CB sliding window program details explained

Following the 2D Laplace equation in Table 9.3, the update of each stencil point for a given time iteration requires summing together the multiplication of each of its surrounding points by a constant coefficient (0.25). Listing 10.2 shows the example of a part of a C-based software program implementation of the 2D Laplace computation, that targets the DRAGON2-CB overlay. Here, a multiplication by the left neighboring point (line 3) is followed by three multiply-and-accumulate operations for each of the surrounding points (of the stencil point being updated), in the remaining directions (line 9 for right neighboring point, line 13 for bottom neighboring point, and line 18,21 or 24 for top neighboring point).

A global index (i) and a pass index (j) are used to track the global position of each point and the corresponding neighbor direction to be used in the stencil update, respectively. In fact, when (j) equals 0, this indicates that the current multiplication operation involves the neighbor point in the left direction. When (j) equals 1,2 or 3, this indicates a multiply-and-accumulate operation that involves the neighbor point that is in the right, down or up directions, respectively.

In every FMUL (multiplication) or FMACCA (multiply-accumulate) operation, the argument RSRC2(.) indicates the address of the neighbor point on the RegisterFile, where (.) can be L for Left, R for Right, D for Down or U for Up. Using the global index (i) of each stencil point that is under update, RSRC2(.) is pre-computed (not shown in Listing 10.2)), to provide the address in the RegisterFile that contains the neighbor point from the (.) direction (L, R, D or U).

Nonetheless, not all surroundings points are stored in the RegisterFile. In fact, the boundary points are stored from previous iterations into the communication buffers. Therefore, the argument OPSRC(.) (where (.) can be L,R,D or U) selects the second operand for the computation (the first is the output of RegisterFile pointed by the address RSRC1). In fact, this argument is pre-computed and selects the second operand from the RegisterFile for neighboring stencil points that are involved in the computation and reside inside the tile boundaries, otherwise, it will select the data operand from the corresponding communication buffer to the current direction.

At every start of a point update, a new point is loaded from LM address (LMADDRLD) (line 4 of Listing 10.2) to the RegisterFile address (RDSTLD), just after the trailing position of the active region shown in Fig. 10.1 (marked by the purple color).

The first FMUL (line 3 of Listing 10.2) and LD (line 4 of Listing 10.2) are actually two instruction packets that are embedded inside the same VLIW instruction, from the view point of micro-architecture implementation, where these two packets target slot1 (Dual Compute Slot) and slot2 (Memory Slot) of the PE, respectively.

The (MODE) instruction field in R-Type instruction may allow storing the result of a computation into the local memory at the location pointed by the content of (LMADDR) field which is then incremented at the end of each of the four-point chunk update (line 33 of Listing 10.2). Here, the (LMADDR) also may loop back to its initial value at every new time iteration to re-update the stencil points, when its limit is reached (sizeLMdata which corresponds to the total number of points in the tile stored into LM). The (MODE) value is pre-computed according to the current (i) position index and is used in the FMACCA operations (line 18,21 and 24 of 10.2).

The lines 35 to 41 in Listing 10.2 show how to manage the global index position of each point (i) and the current direction of the neighbor point involved in the stencil update (j), which later select the corresponding operation to be performed for the update of a given stencil point. Here, when the index of the following point indicated by (i+1) belongs to the contiguous range of the four points under update (line 40 of Listing 10.2), the direction of the surrounding neighbor point involved in the update operation that is indicated by index (j) remains unchanged. Otherwise, two different cases may be encountered (line 35 of Listing 10.2). The first case happens when the update of the points in the current four contiguous points is not fully completed for the given pass (line 36 of Listing 10.2). In this case, the index (j) is incremented (line 37 of Listing 10.2) to perform a new multiplication and accumulation (FMACCA) operation using the next direction of the surrounding points of the current stencil point that is under update. Moreover, the index (i) is reset to the position of the first point in the four-point chunk (line 38 of Listing 10.2), to start a new pass of multiplication and accumulation on the next neighbor points direction. The other case is when all of the points in the contiguous four-point chunk have been completely updated for the given pass. Here, the index (j) is reset to zero and a new set of four contiguous points will be updated after the increment of the index (i) (line 39 of Listing 10.2). Ultimately, the arguments BROFFSET and BMADDR are unused in the FMUL and FMACCA instructions from the program shown in Listing 10.2.

10.3 Comparison with related works

Table 10.2: Comparison of the **Double-Precision** Sustained Performance, Power Efficiency and the EPR with other related works [4].

	Ref	[9]	[9]	[10]	[125]	Ours$^{\text{OMP}}$	**Ours**
	Year	2016	2016	2019	2012	2022	2022
	Type	FPGA	FPGA	FPGA	GPU	CPU	FPGA
	Device	DE5	395-D8	Nallatech385	GTX580	Core i9 9900K	Alveo U280
2D Laplace	**Perf.** [GFLOPS]	-	-	115	-	50.91	**135.79**
	P.Eff [GFLOPS/W]	-	-	-	-	0.53	**4.20**
	EPR [%]	-	-	-	-	22.09	**87.31**
2D Jacobi	**Perf.** [GFLOPS]	27.3	40.9	104	49.5	66.87	**139.72**
	P.Eff [GFLOPS/W]	-	-	-	-	0.7	**4.33**
	EPR[%]	-	-	-	-	29.02	**89.84**
3D Jacobi	**Perf.** [GFLOPS]	27.2	40.7	74	50	43.4	**145.62**
	P.Eff [GFLOPS/W]	-	-	-	-	0.45	**4.24**
	EPR[%]	-	-	-	-	18.83	**91.93**
4D Jacobi	**Perf.** [GFLOPS]	-	-	-	-	50.2	**105.77**
	P.Eff [GFLOPS/W]	-	-	-	-	0.52	**3.64**
	EPR[%]	-	-	-	-	21.78	**91.81**

$^{\text{OMP}}$ Programs use OpenMP and are compiled using g++(7.5.0) (with -fopenmp -O3 -mavx2).

This section provides a quantitative comparative study with related state-of-the-art works. The work in this thesis mainly targets double-precision computations. Nonetheless, a single-precision implementation for the 288-PE DRAGON2-CB overlay was investigated as well. This was achieved by splitting the PE data-path into two parallel 32-bit ALUs and FPUs using the same RegisterFile width that packs two 32-bit data in every 64-bit register location. All other aspects of the overlay architecture are maintained the same in both precisions. The single-precision DRAGON2-CB overlay operates at 275 MHz with the 2D-Mesh interconnect, 270 MHz with the 3D-Mesh interconnect and 240 MHz with the 4D-Mesh interconnect.

This section reports and discusses some of the previous works that are most relevant to the work in this thesis. The related results are reported in Table 10.2 (double-precision results) and Table 10.3 (single-precision results) and cover aspects such as the computational performance, the power efficiency as well as the Effective-to-peak Performance Ratio (EPR). In addition, these two tables include the results obtained with the 288-PE DRAGON2-CB implementations. Here, an additional digit of precision is revealed into the DRAGON obtained results that were reported in Table 8 and 9 in [4] with only one digit precision after the decimal point, to provide a uniform precision with two digits after the decimal point. Moreover, Table 10.2 and Table 10.3 report results obtained using a single chip, be it a FPGA, a CPU or a GPU, for a fair comparison.

10.3.1 FPGA-based works

The work in reference [51] proposes a streaming and programmable architecture specifically designed for solving Jacobi-like stencil computations. While the proposed architecture is scalable on multi FPGAs, only the obtained results for a single-FPGA are reported in Table 10.3.

The PE architecture in reference [51] embeds one constant memory and another addressable memory. The constant memory is used for storing the coefficients involved in the computation of each stencil point. On the other hand, the addressable memory acts as a buffer that is used to exchange data with neighboring PEs, or to store the results of computation for a given iteration update. The data-path of this PE contains eight-stage pipeline with a total of five computational stages dedicated for the multiplication and accumulation operations. These operations are performed thanks to the FMAC (Floating-Point Multiply-ACcumulate Unit) unit. The design of this unit has inspired the architecture of the proposed MAC FPU (Multiply-ACcumulate Floating-Point Unit) in this thesis, which offers additional instructions and targets 64-bit double-precision instead of 32-bit single-precision floats. Therefore, the proposed MAC FPU micro-architecture in the DRAGON2 and DRAGON2-CB overlays, has an eight-stage pipeline from which four stages are dedicated to the accumulation step, in contrast to the two stages of accumulation that were implemented in the FMAC of [51]. The additional stages aim to improve timing outcome by reducing the increased amount of logic levels in the 64-bit version.

The work in reference [126] proposed a stencil calculation acceleration approach that consists of a queue of streaming computational blocks that are called SST (Streaming Stencil Timestep). While, the proposed method achieves a nearly linear speedup alongside an increase in the amount of implemented SSTs in 2D and 3D Jacobi benchmarks, its scalability seems to be limited by the benchmark dimension. In fact, the BRAM memory resource seems to be the main cause that is limiting scalability for a 3D Jacobi calculation, as shown by the related resource utilization report in [126]. In contrast, the DRAGON2-CB overlay adopts an efficient compact buffering scheme that efficiently uses BRAM resources, which leads to a good overall scalability. As a result, a similar amount of 288 PEs was implemented with either a 2D or 3D interconnects, which has led the 2D and the 3D Jacobi benchmarks to achieve comparable performance results on the 2D and 3D interconnect versions of the DRAGON2-CB, respectively.

Table 10.3: Comparison of the **Single-Precision** Sustained Performance, Power Efficiency and the EPR with other related works [4].

	Ref	[51]	[126]	[9]	[9]	[127]	[10]	[128]	[9]	[123]	[9]	[129]	[129]	**Ours**
	Year	2014	2015	2016	2016	2018	2019	2021	2016	2018	2016	2018	2018	2022
	Type	FPGA	FPGA	FPGA	FPGA	FPGA	FPGA	FPGA	CPU	CPU	GPU	GPU	GPU	FPGA
	Device	EP3SL 150	XC7VX 485T	DE5	395-D8	ADM-PCIE-KU3	Nallatech 385	XC7VX 485T	Core i7 4960X	Xeon E5-2630	GTX 960	Tesla P100	Tesla V100	Alveo U280
2D Laplace	**Perf.** [GFLOPS]	-	-	181.9	175.7	-	**659**	-	31.7	-	73.8	-	-	276.61
	P.Eff [GFLOPS/W]	-	-	-	-	-	**13.1**	-	-	-	-	-	-	8.77
	EPR [%]	-	-	**92.8**^{i}	11.7^{i}	-	48.2^{A}	-	18.3^{C}	-	3.2	-	-	87.31
2D Jacobi	**Perf.** [GFLOPS]	34	23.6	133.3	237.8	90.04	**763**	160.81	65.9	-	164.1	-	-	284.62
	P.Eff [GFLOPS/W]	0.8	-	-	-	-	**15.1**	7.1	-	-	-	-	-	8.82
	EPR[%]	87.4	-	68^{i}	15.8^{i}	-	55.8^{A}	-	38.1^{C}	-	7.1	-	-	**89.84**
3D Jacobi	**Perf.** [GFLOPS]	31.9	2.63	111.3	193.3	83.98	628	66.06	-	-	-	1205.3	**2111**	285.96
	P.Eff [GFLOPS/W]	0.71	-	-	-	-	**11**	3.33	-	-	-	6.4	8.1	8.50
	EPR[%]	83.9	-	59.3^{i}	12.8^{i}	-	45.9^{A}	-	-	-	-	12.9^{P}	13.4^{V}	**91.93**
4D Jacobi	**Perf.** [GFLOPS]	-	-	-	-	-	-	-	-	70.65	-	-	-	**253.85**
	P.Eff [GFLOPS/W]	-	-	-	-	-	-	-	-	0.74	-	-	-	**7.79**
	EPR[%]	-	-	-	-	-	-	-	-	64.22^{X}	-	-	-	**91.81**

^{C}I recalculated and updated the EPR value given in [9] based on the official Intel data [130] that provides the peak performance of the Core i7-4960X.
^{i}The reported EPR is based on a TPP that corresponds to what the authors claim as the peak performance of the underlying FPGA.
^{A}EPR is computed in the same way as the authors previous work in [9] and thus it is based on a TPP that corresponds to the peak performance of the underlying FPGA which can be retrieved in [131].
^{X}EPR is computed based on the official Intel data [132] that provides the peak performance of the Xeon E5-2630.
^{P}EPR is computed based on the official Nvidia data [133] that provides the peak performance of the Tesla P100 PCI-E.
^{V}EPR is computed based on the official Nvidia data [134] that provides the peak performance of the Tesla V100 SXM2.

The work in reference [9] proposes a streaming architecture that is based on an OpenCL programming approach, to accelerate stencil computing models. It consists of chained compute blocks called PCMs (Pipelined Computing Module). These PCMs consist as well of several PEs accepting input data in a cascaded manner thanks to shift-registers. Here, the amount of PCM modules is equal to the amount of stencil iterations that are computed in a parallel manner. The DRAM memory is equivalent to a global memory and is connected only to the first and last PCMs in the cascade, to provide the input of non-processed data and store back the completed result of processed data, respectively. This approach achieved impressive results peaking at 237.8 GFLOP/s for a single-precision computational performance on a 2D Jacobi stencil benchmark. On a 2D Laplace stencil benchmark with the same precision it managed to hit an EPR of 92.8%. Using a double-precision implementation, the obtained results were significantly reduced as can be seen in Table 10.2. While the DSP-based hardened multipliers were used for the multiplication operations, ALMs (Adaptive Logic Module) were used to implement the addition operations, as claimed by the authors in reference [9]. Therefore, the increased size of exponent and mantissas in double-precision quickly becomes an area bottleneck due to the significant use of ALMs in such an implementation. Moreover, The DSPs of the Stratix V FPGA [135] contained in the DE395-D8 (1963 DSPs) and DE5 (256 DSPs) boards, have multipliers capable of performing multiplication operations with variable precisions with inputs widths up to 27x27. Hence, a single-precision multiplication of two 23-bit mantissas would be performed by a single DSP multiplier, whereas a double-precision multiplication of two 52-bit mantissas would either be implemented by multiple DSPs, or otherwise, a single DSP that takes multiple clock cycles for computing the partial products. Consequently, this may explain the performance drop using double-precision floating-point data.

Nonetheless, despite the fact that the work in reference [9] adopts a specifically-tailored hardware implementation to compute stencil models, it is still significantly outperformed by the proposed overlay architecture in this thesis, that manages to achieve better sustained performance in 2D and 3D Jacobi stencil benchmarks, as can be seen in 10.2 (double-precision results) and Table 10.3 (single-precision results).

An apparently newer version of the work in [9] is proposed in reference [10] and is implemented on a larger FPGA (Arria 10 GX1150 [131]) with an enhanced implementation as compared to the originally proposed architecture. Here, the work in reference [10] benefits from the increased peak performance that reaches 1366 GFLOPS which allowed the sustained performance to be increased in a single-precision 2D Jacobi stencil benchmark to reach 763 GFLOPS. This jump in performance is mainly due to the availability of DSPs that can implement hardened single-precision floating-point additions and multiplications. While these hardened DSPs allowed the work in [10] to outperform the sustained performance obtained by the overlay architecture proposed in this thesis, the lack of support for double-precision operations caused the related performance in [10] to be locked at a significantly lower performance of 104 GFLOPS.

Besides, SODA was another work proposed in [127] that consists of an automated framework targeting FPGA implementation of stencil calculations. SODA generates data-flow optimized architectures from a high-level parameter description of a stencil computing model. As a result, it outputs FPGA-based HLS kernels in C++ language and a host-based API (Application Programming Interface) based on OpenCL (Open Computing Language). While the proposed overlays in this thesis adopt a similar programming model split into an OpenCL-based host and an FPGA-based kernel, they provide a more optimized HDL-based kernel implementation that is hand-tuned through guided placement and manual physical floorplan, in addition to hand-picked implementation strategies. Moreover the proposed kernel in this thesis that is based on a software-programmable overlay architecture, provides higher flexibility and a better re-usability scheme thanks to the possibility of addressing various computing models by simply downloading new program instructions while maintaining the same kernel bitstream file.

The work proposed in reference [128] investigated the efficient FPGA-based stencil computing implementations, through a library of components that are designed in HDL and offer the possibility of customization. The proposed stencil accelerator design consists of serially chained computational blocks (named SST). While the proposed approach to accelerate the stencil computations consists of cascaded computing blocks that seem to offer a good scalability, the related resource utilization increases alongside an increase in the number of iterations to update the stencil points.

In contrast to this undesired effect, the DRAGON architecture, across all of its micro-architecture implementations, maintains a fixed resource utilization when increasing the number of stencil update iterations, thanks to its software aspect that allows managing loops with the help of dedicated instructions.

In addition, the performance results provided in [128] were normalized as compared to a multi-thread CPU reference. To obtain the absolute numbers of these results, one of the normalized reported values have been compared with its original performance value that was given by the work in [51]. The deduced corresponding single FPGA result for the work in [128] is reported in Table 10.3.

10.3.2 CPU-based works

The work in reference [123] proposes a domain-specific language that allows generating optimized and high-performance code for stencil computations from higher level abstractions. The performance outcomes of such an approach were investigated in [123] for stencil benchmarks with multiple dimensions. For example, a single CPU node achieves a performance of about 7 Giga stencil/s (as was approximately deduced from a given performance graph), for a 4D Jacobi stencil benchmark based on single-precision floats.

Based on the number of operations required to update each stencil point, which the authors in [123] claim to be the sum of eight additions and two multiplications, the corresponding performance in [GFLOPS] has been computed and reported in Table 10.3.

While a direct comparison with this implementation is difficult because of the lack of sufficient implementation details, the related results are reported here to serve as reference for CPU-based implementations.

10.3.3 GPU-based works

The DRAGON2-CB overlay achieves better computational performance when compared to GPUs from previous generations, namely, the GTX960 and GTX580 GPUs. For example, the GTX960 GPU has a peak performance of around 2.3 TFLOPS using 32-bit single-precision floats [9]. However, it provides a small fraction of this peak when it comes to 64-bit double-precision computations as its original peak performance is divided by a factor of 32 due to the ratio of double-precision cores that amount to 1/32 of the single-precision cores, which leads to a peak double-precision performance of about 72 GFLOPS.

Nonetheless, GPU sizes has been growing at a significant rate due to the advances in technology manufacturing process and the reduction of transistor sizes. As a result, the computational performance of these devices has been increased dramatically for single- and double-precision, but so did their power dissipation. An example is given by the work in [129] which reports the power efficiency and sustained performance results for the NVidia V100 and P100 GPUs using single-precision floats.

Knowing that the double-precision peak performance of these two GPUs are nearly half of those achieved by single-precision floats, and with the assumption that the same EPR can be perfectly achieved in double-precision computations, the reported results in [129] for the power efficiency and the sustained performance of these devices will be simply divided by two. This would lead to a performance of 602.65 GFLOPS and 1055.5 GFLOPS and a power efficiency of 3.2 GFLOPS/W and 4.05 GFLOPS/W, for P100 and V100 GPUs, respectively. While these results still outperform the DRAGON2-CB in terms of sustained performance, the power efficiency results can be slightly improved.

10.4 Summary

In this chapter, I presented a summary of differences between the provided three micro-architecture implementations of this work, namely, the Baseline DRAGON, DRAGON2 and DRAGON2-CB micro-architectures. I reported the best achieved results (for the largest size implementation of these overlays) and discussed the various impacts of the micro-architecture enhancements. Ultimately, I provided a comparison with other state-of-the-art related works where I discussed details of their corresponding implementations and achieved results.

Part V

Conclusion

Chapter 11

Conclusion

Reconfigurable chips such as FPGAs offer a convenient solution for energy-efficient computing, thanks to their flexibility to implement customized circuits and interconnections which reduces the impact of data movement on energy consumption. Despite this appealing advantage, these chips remain difficult to program or use with a host-based platform and many software programmers find themselves unable to efficiently and easily harness the benefits of such devices. Furthermore, the abstraction of an FPGA using an overlay architecture comes with significant overhead costs that often negatively impact its computational performance, energy efficiency and resource utilization.

The work in this thesis proposes an **energy-efficient many-core overlay architecture for reconfigurable chips** that aims to solve these issues. Here, a summary of the achieved contributions is presented along with the expected societal impacts and examples of possible applications that may benefit from the work proposed in this thesis.

11.1 A summary of achievements and contributions

11.1.1 General contributions and achievements

To address the general research objectives presented earlier, the work in this thesis proposes the DRAGON overlay architecture. A summary of the major contributions achieved thanks to the proposed overlay architecture are presented as follows:

- The proposed overlay architecture adopts a custom-design ISA that provides a software programmability layer on top of the FPGA fabric to bridge the gap with millions of software programmers. This layer also facilitates the control and integration within a heterogeneous host-based computing platform. Furthermore, a novel general methodology to use programmable overlay architectures within this kind of platforms is proposed by leveraging and extending the low-level infrastructure provided by the FPGA vendor. This allows the overlay to be abstracted as a standard OpenCL task which provides a simple yet efficient model to control and use FPGA devices for the acceleration of computations.

- The proposed overlay achieves high levels of computational performance and energy-efficiency as compared to the state-of-the-art hardware-specific implementations of similar applications. The high computational performance and energy efficiency levels are achieved without compromising re-usability. That is the proposed overlay can be reprogrammed, through different software programs to address various computing applications without the need of redesigning the underlying circuit.

- To extract the highest levels of computational performance and energy efficiency from the target FPGA while minimizing the overhead costs intrinsically related to overlays, the proposed work in this thesis implements a custom-design ISA and leverages multiple existing parallel processing paradigms such as SIMD, VLIW and DAE and combines them to create a unique highly-efficient many-core-processor FPGA overlay architecture. As a result, for the same benchmarks, the proposed overlay (for example with the DRAGON2-CB micro-architecture) outperformed all of the previous double-precision-based implementations, using a single chip. It also outperformed all previous single-chip, single-precision implementations that target Xilinx FPGAs.

11.1.2 Detailed contributions and achievements

A preliminary micro-architecture named the Baseline DRAGON overlay has been introduced and its FPGA implementation has been carefully investigated to study the various issues facing the achievement of the different research objectives. While the Baseline DRAGON achieved an impressive EPR that is over 87% for a complex stencil computing application, simply relying on the default settings of the FPGA compiler led to an implementation that unfairly capped the performance and energy-efficiency potentials of DRAGON.

Issues related to achieving higher clock speed, scaling the overlay size and the degree of the interconnect, or improving the energy efficiency, have been investigated, identified, and then, separate solutions have been proposed and implemented through two enhanced micro-architecture implementations named DRAGON2 and DRAGON2-CB.

The first identified issue was the scarcity of inter-die wires called SLLs. Since the target FPGA (ALVEO U280) has all its HBM2 memory banks located into a single SLR region, moving data between the Global Memory (residing in HBM2 banks) and the Broadcast Memories (residing in different SLRs), required the use of a significant amount of SLLs. Therefore, first I targeted a model where the BC distribution is balanced across SLRs. Then, I provided a mathematical formulation that is based on the physical layout configuration of BCs, in the underlying overlay implementation, to obtain the optimal AXI data bus width, that maximizes the utilization of SLLs, with respect to the limited availability of inter-die wires. As a result, I found that a data bus not exceeding 256 bits in width, leads to an improved scalability of both the overlay size and the interconnect degree, of the implemented DRAGON2 and DRAGON2-CB overlays.

The second identified issue is the significant amount of BRAMs that are required to implement the communication buffers, particularly, for larger degrees of the interconnect. As such, I introduced a buffering model called Compact Buffering, that consists of using one BRAM to host circular buffers on the 2D space and gradually increasing the amount of BRAMs for each added dimension, while sharing the memory space inside each BRAM between the multiple input directions, to store incoming data from the different neighboring PEs with support from the underlying ISA. This buffering model aims at efficiently using BRAM resources to improve scalability and provide the ability of extending the interconnect degree to higher dimensions while nearly maintaining the same EPR. Consequently, the BRAM utilization was reduced to nearly 50%, and more PEs were deployed using this scheme, particularly, in the 4D version of the interconnect, effectively boosting the DRAGON overlay scalability, and improving its overall performance and energy-efficiency.

Besides, the clock speed was a limiting factor of performance and power-efficiency. Therefore, I introduced deeper pipelines in the PE, MAC FPU and AXI data buses, which led to mostly doubling the original clock speed (130MHz to more than 270 MHz in most configurations) and improving the energy efficiency in the overall design.

The combined micro-architecture and technology-related improvements that were proposed and implemented, have led to around 2x in performance increase and 3x in power-efficiency improvement over the Baseline DRAGON overlay, with the exact degree of the interconnect (2D) and the exact amount of PEs (144). Moreover, the improved scalability has led to deploying more PEs which resulted in around 4x improvements in the overall performance and power-efficiency over the Baseline DRAGON overlay.

Ultimately, the enhanced DRAGON2-CB that leverages the Compact Buffering technique, remarkably achieves, at best, EPRs of 89.84%, 91.93% and 91.81%, in 2D, 3D and 4D Jacobi benchmarks, respectively. The corresponding double-precision performance reaches 139.72 GFLOPS, 145.62 GFLOPS and 105.77 GFLOPS, while the power-efficiency achieves more than 4 GFLOPS/W, in double-precision and more than 8 GFLOPS/W in single-precision, in 2D and 3D interconnects with a slight drop in the 4D version due to the decreased operating clock speed that reduced the computational performance.

11.2 Benefits to the community and examples of application domains

11.2.1 Summary of qualitative results

In summary, the proposed many-core overlay architecture achieves impressive EPR figures, comparable energy efficiency results to the high-end GPUs and relatively high computational performance as compared to the previous FPGA-based dedicated designs and overlays. The proposed architecture has been proven to scale really well alongside an increase in compute resources. As such, the computational performance has shown a near linear speedup. In addition, the achieved power-efficiency has shown a considerable improvement as well. Moreover the EPR remained almost constant when increasing the amount of PEs which proves that the architecture scales well and maintains the same performance ratio to its theoretical peak in larger design sizes.

11.2.2 FPGA-based accelerator for heterogeneous computing

The DRAGON architecture achieves impressive results that are only capped by the size of a single FPGA device. This may be overcome with larger FPGA sizes or the expansion of the underlying architecture to clusters of multiple FPGAs, based on the proven DRAGON scalability. As such, DRAGON may become the base building block of next generation accelerators based on reconfigurable devices. Besides, the proposed OpenCL-based control of the many-core overlay architecture, makes it even easier to achieve this goal, by offering a GPU-like model of integration within a heterogeneous computing platform. Furthermore, the proposed ISA in this work is designed with the possibility of extension making it an attractive approach to maintain and upgrade the capabilities of such an accelerator in the future systems. Besides, the proposed overlay provides comparable or even higher energy efficiency compared to dedicated solutions without sacrificing its flexibility and hence it contributes to reduce the impact of higher energy consumption on the environment, making it an attractive alternative for green computing.

11.2.3 ASIC possibility with uncapped capabilities

While the proposed architecture is primarily destined for reconfigurable chips such as FPGA, the underlying ISA and the proposed micro-architectures in this work, may be used to tape-out a full-fledged ASIC device, where the area and speed limitations are more controllable and less capped. As a result, the raw computational performance of GPUs may be matched while the high EPR that is related to the architecture and that is nearing 90% across several benchmarks may push the sustained performance farther than traditional accelerators, hence, unlocking its real potential. Moreover, it remains possible to specialize the proposed overlay towards a particular kind of application through micro-architectural enhancements, such as adopting novel interconnect topologies, or through the architecture itself thanks to the possibility of ISA extensions, effectively allowing DRAGON to become a vessel for hardware and software design space exploration towards future research efforts on building high-performance and energy-efficient architectures.

11.2.4 Bridging the gap with millions of software users

In the future, the proposed overlay architecture as well as its seamless integration methodology into a heterogeneous computing platform, aims to bring millions of software programmers closer to FPGAs and allow them to harness its capabilities without facing the traditional challenges that a hardware expert may encounter during the programming or the control of such devices.

11.2.5 Paving the way towards new discoveries

The experimental evaluation based on a set of stencil benchmarks showed promising results which implies that DRAGON may be used in scientific applications that are based on these kinds of stencil calculations, such as the numerical simulation of the Poisson equation (which is a generalized form of the Laplace equation) or the diffusion equation. For example, the ISA support for high degrees of interconnects provides a solid means to eventually address high-dimensional problems such as the classical 3D wave equation [136] or the relativistic 4D wave equation [137], in the four-dimensional space-time of Minkowski [138]. As such, the proposed overlay may be used for the simulation of physical phenomena with applications in astrophysics, electrodynamics and computational fluid dynamics which may pave the way towards new discoveries.

11.2.6 Towards a highly-efficient AI accelerator

The high EPR figures of the proposed architectures in stencil based benchmarks shows a promise for applications in AI (Artificial Intelligence) where a high sustained performance may be maintained near the theoretical peak resulting in a very efficient computing accelerator. Basically, AI applications such as CNN, are split into two major computations, convolutions and matrix multiplications. The convolution operations exhibit a similar computational pattern as stencil applications which hints that DRAGON may be investigated to target such kind of applications. Besides, the broadcasting feature of DRAGON is inherited from the EXACC[8] architecture which had the acceleration of matrix multiplication operations in sight. As such, it may perform such computations with a reduced bandwidth requirement, since a single data can be broadcasted to multiple PEs to perform a multiplication or a multiply-and-accumulate operations backed by the MAC FPU. The architectural support of these two key CNN operations (convolution and matrix multiplication) makes DRAGON an attractive architecture to investigate the acceleration of such kind of workloads.

Annex 1: Assembly Example on the DRAGON2-CB: 2D Laplace benchmark (C program embedding C-based DRAGON ISA assembly opcode functions)

```

#include <stdlib.h>
#include "functions.h"
#include <vector>
#include "time.h"
#include <cstdint>
#include <iostream>
#include <sstream>
#include <fstream>

using namespace std;

//number of stencil points in one dimension of the tile
#define data_line 32
//total number of points in one tile (to be stored in LM)
#define data_count data_line*data_line

#define READ_LATENCY 150
#define WRITE_LATENCY 150


unsigned long NDST_UP             =   0b00000001; //=> write to up direction
unsigned long NDST_LEFT           =   0b00000010; //=> write to left direction
unsigned long NDST_RIGHT          =   0b00000100; //=> write to right direction
unsigned long NDST_DOWN           =   0b00001000; //=> write to down direction
unsigned long NDST_UP_and_LEFT    =   0b00000011; //=> write to up and left direction
unsigned long NDST_UP_and_RIGHT   =   0b00000101; //=> write to up and right direction
unsigned long NDST_DOWN_and_LEFT  =   0b00001010; //=> write to down and left direction
unsigned long NDST_DOWN_and_RIGHT =   0b00001100; //=> write to down and right direction

unsigned long OPCODE      = 0x0;
unsigned long RSRC1       = 0x0;
unsigned long MODE        = 0x0;
unsigned long UNUSED      = 0x0;
unsigned long BROFFSET    = 0x0;
unsigned long BMADDR      = 0x0;
unsigned long RSRC2       = 0x0;
unsigned long RDST        = 0x0;
unsigned long OPSRC       = 0x0;
unsigned long IMMEDIATEMSB= 0x0;
unsigned long IMMEDIATELSB= 0x0;
unsigned long LMADDR      = 0x0;
unsigned long NDST        = 0x0;
unsigned long MASKLOAD    = 0x0;
unsigned long DATACOUNT   = 0x0;
unsigned long NSRC        = 0x0;
unsigned long FUNCT       = 0x0;
unsigned long ITERATIONS  = 0x0;
unsigned long BURSTSIZE   = 0x0;
unsigned long BMOFFSET    = 0x0;
unsigned long GMOFFSETMSB = 0x0;
unsigned long GMOFFSETLSB = 0x0;
unsigned long INSTRUCTION = 0x0;
unsigned int  INSTR_LSB   = 0x0;
unsigned int  INSTR_MSB   = 0x0;


unsigned long LMADDRLD    = 0x0;
unsigned long RDSTLD    = 0x0;
unsigned long RSRC2U    = 0x0;
unsigned long RSRC2L    = 0x0;
unsigned long RSRC2R    = 0x0;
unsigned long RSRC2D    = 0x0;

int main(int argc, char **argv){
  unsigned int iters;
  iters = atoi(argv[1]);
  string itersize = "_"+std::to_string(iters)+"iter";
  string dimsize = std::to_string(data_line)+"x"+std::to_string(data_line);
  //executable file that contains instructions that will be stored in IM
  ofstream f;
  string filename = "2Dlaplace"+dimsize+itersize+".txt";
  f.open(filename, ios::out);
  //-----------------------------------------------------
  //    program start
  //-----------------------------------------------------
  NOP(f);
  Bflush(0xFF,f);
  //-----------------------------------------------------
  //load constant in register file addr 255,
  //value 0.25 = 0x3FD0 0000 0000 0000
  IMMEDIATEMSB = 0x3FD0;
  IMMEDIATELSB = 0x000000000000;
  RDST = 255;
  LDimm(IMMEDIATEMSB, IMMEDIATELSB, RDST, f); //equivalent to 2 slots
  //-----------------------------------------------------
  //move an array of 16 32x32 64bit data from GM to BMs (16 PEs per BC)
  //-----------------------------------------------------
  BURSTSIZE   = 128;
  BMOFFSET    = 0;
  GMOFFSETMSB = 0;
  GMOFFSETLSB = 0;
  for (int i=0; i<32;i++){
    RDGMEM(BURSTSIZE,BMOFFSET,GMOFFSETMSB,f);
    RDGMEM(0,0,GMOFFSETLSB,f);
    //-----------------------------------------------------
    ITERATIONS  = READ_LATENCY; //cycles to complete transaction
    REPEAT(ITERATIONS,f);
    NOP(f);
    //-----------------------------------------------------
    BNZ(f);
    NOP(f);
    //-----------------------------------------------------
    NOP(f);
    NOP(f);
    BMOFFSET    = BMOFFSET+32;
    GMOFFSETLSB = GMOFFSETLSB+128*32;  //128 beats * 32 Bytes (4096Bytes)
  }
  //the cycles required are finished
  //-----------------------------------------------------
  //-----------------------------------------------------
```

```
// load  data from BM to LM
MASKLOAD    = 0xF0;  //load data from PE0 to PE15
MODE        = 0;     //no broadcast
LMADDR      = 0;
BROFFSET    = 0;
BMADDR      = 0;
DATACOUNT   = data_count-1;
NOP(f);
LDBM(MASKLOAD,MODE,LMADDR,BROFFSET,BMADDR,DATACOUNT,f);
//----------------------------------------------------
//----------------------------------------------------
//the cycles required for RDGMEM are finished
//----------------------------------------------------
ITERATIONS  = 1024; //cycles to complete transaction
REPEAT(ITERATIONS,f);
NOP(f);
//----------------------------------------------------
BNZ(f);
NOP(f);
//----------------------------------------------------
NOP(f);
NOP(f);
//----------------------------------------------------
//the cycles required for LDBM are finished
//----------------------------------------------------
//------------------------------------------------------------------------------------------
//----------------------------------------------------
//scatter first row of data to up direction
//----------------------------------------------------
MODE = 0b10;  //scatter from LM
RSRC2 = 0x0; //not used here
NSRC  = 0x0; //not used here
NDST  = 0b00000001; // write to up direction
LMADDR= 0x0; //base address
for (int i=0; i<data_line;i++){
  NOP(f);
  NSG( MODE, LMADDR, RSRC2, NDST, NSRC, f);
  LMADDR++;
}
//----------------------------------------------------
//scatter last row of data to down direction
//----------------------------------------------------
MODE = 0b10;  //scatter from LM
RSRC2 = 0x0; //not used here
NSRC  = 0x0; //not used here
NDST  = 0b00001000; // write to down direction
LMADDR= data_line*(data_line-1); //base address
for (int i=0; i<data_line;i++){
  NOP(f);
  NSG( MODE, LMADDR, RSRC2, NDST, NSRC, f);
  LMADDR++;
}
//----------------------------------------------------
//scatter first col of data to left direction
//----------------------------------------------------
MODE = 0b10;  //scatter from LM
RSRC2 = 0x0; //not used here
NSRC  = 0x0; //not used here
NDST  = 0b00000010; // write to left direction
LMADDR= 0; //base address
for (int i=0; i<data_line;i++){
  NOP(f);
  NSG( MODE, LMADDR, RSRC2, NDST, NSRC, f);
  LMADDR+=data_line;
}
//----------------------------------------------------
//scatter last col of data to right direction
//----------------------------------------------------
MODE = 0b10;  //scatter from LM
RSRC2 = 0x0; //not used here
NSRC  = 0x0; //not used here
NDST  = 0b00000100; // write to right direction
LMADDR= data_line-1; //base address
for (int i=0; i<data_line;i++){
  NOP(f);
  NSG( MODE, LMADDR, RSRC2, NDST, NSRC, f);
  LMADDR+=data_line;
}
//--------------------------------------------------
//boundaries are now scattered to buffers
//--------------------------------------------------
// load 2 line of data from LM to regfile
LMADDR      = 0;
RDST        = 0;
for (int i=0; i<(2*data_line);i++){
  NOP(f);
  LD(LMADDR, RDST, f);
  LMADDR++;
  RDST++;
}

for (int i=0; i<16;i++){ //15 cycles latency
  NOP(f);
  NOP(f);
}
//
//--------------------------------------------------
//----------------------------------------------------
//start updating and iterating the stencils
//--------------------------------------------------
//1st stencil corner top left to 64th stencil bottom right
//--------------------------------------------------
//--------------------------------------------------
REPEAT(iters,f);
NOP(f);
//--------------------------------------------------
RSRC1 = 255; //addr for constant 0.25
BROFFSET = 0x0; //not used here
BMADDR =0x0; //not used here
RDST = 0;  //not used here
NDST = 0b00000011; //scatter to up and left dirs
unsigned long sizeLMdata = data_line*data_line;
unsigned long sizeRFdata = data_line*4;

unsigned long OPSRCU = 0;
unsigned long OPSRCR = 0;
unsigned long OPSRCD = 0;
unsigned long OPSRCL = 0;

RSRC2U = 0;
RSRC2R = 1;
RSRC2D = data_line;
RSRC2L = 0;

LMADDR = 0;
LMADDRLD = 2*data_line;
RDSTLD = 2*data_line;

int i = 0;
int j = 0;

while (i< sizeLMdata){


  RSRC2R = ((i%sizeRFdata)+1) & 0xFF;
  RSRC2L = ((i%sizeRFdata)-1) & 0xFF;

  if ((i%sizeRFdata)>=data_line) {
      RSRC2U = ((i%sizeRFdata)-data_line) & 0xFF;
  } else {
      RSRC2U = ((i%sizeRFdata)+(3*data_line)) & 0xFF;
  }

  if ((i%sizeRFdata)<=((3*data_line)-1)){
      RSRC2D = ((i%sizeRFdata)+data_line) & 0xFF;
  } else {
      RSRC2D = ((i%sizeRFdata)-(3*data_line)) & 0xFF;
  }

  // select destination buffer if in boundaries
  if (i==0){       // TOP LEFT CORNER
    MODE = 0b11;  //store to LM and broadcast to neighbors
    NDST   = NDST_UP_and_LEFT;
    OPSRCU = 0b0011;
    OPSRCR = 0b0000;
    OPSRCD = 0b0000;
    OPSRCL = 0b0100;
  } else if ((i > 0) && (i < (data_line-1))) { // TOP ROW WITHOUT CORNERS
    MODE = 0b11;  //store to LM and broadcast to neighbors
    NDST = NDST_UP;
    OPSRCU = 0b0011;
    OPSRCR = 0b0000;
    OPSRCD = 0b0000;
    OPSRCL = 0b0000;
  } else if (i==(data_line-1)) {  // TOP RIGHT CORNER
    MODE = 0b11;  //store to LM and broadcast to neighbors
    NDST = NDST_UP_and_RIGHT;
    OPSRCU = 0b0011;
    OPSRCR = 0b0101;
    OPSRCD = 0b0000;
    OPSRCL = 0b0000;
  } else if ((i % data_line) == 0 && (i !=(data_line*(data_line-1)))) {  // LEFT COLUMN
    MODE = 0b11;  //store to LM and broadcast to neighbors
    NDST = NDST_LEFT;
    OPSRCU = 0b0000;
    OPSRCR = 0b0000;
    OPSRCD = 0b0000;
    OPSRCL = 0b0100;
  } else if (((i+1) % data_line) == 0 && (i < ((data_line*data_line)-1)) ) {  // RIGHT COLUMN
    MODE = 0b11;  //store to LM and broadcast to neighbors
    NDST = NDST_RIGHT;
    OPSRCU = 0b0000;
    OPSRCR = 0b0101;
    OPSRCD = 0b0000;
    OPSRCL = 0b0000;
  } else if (i==(data_line*(data_line-1))) {  // BOTTOM LEFT CORNER
    MODE = 0b11;  //store to LM and broadcast to neighbors
    NDST = NDST_DOWN_and_LEFT;
    OPSRCU = 0b0000;
    OPSRCR = 0b0000;
    OPSRCD = 0b0110;
    OPSRCL = 0b0100;
  } else if ((i > (data_line*(data_line-1))) && (i < ((data_line*data_line)-1))) { //BOTTOM ROW WITHOUT
↪   CORNERS
    MODE = 0b11;  //store to LM and broadcast to neighbors
    NDST = NDST_DOWN;
    OPSRCU = 0b0000;
    OPSRCR = 0b0000;
    OPSRCD = 0b0110;
    OPSRCL = 0b0000;
  } else if (i==((data_line*data_line)-1)) { //BOTTOM RIGHT CORNER
    MODE = 0b11;  //store to LM and broadcast to neighbors
    NDST = NDST_DOWN_and_RIGHT;
    OPSRCU = 0b0000;
    OPSRCR = 0b0101;
    OPSRCD = 0b0110;
    OPSRCL = 0b0000;
  } else {  //internal data
    MODE = 0b01;  //store to LM
    NDST = 0b00000000;
    OPSRCU = 0b0000;
    OPSRCR = 0b0000;
    OPSRCD = 0b0000;
    OPSRCL = 0b0000;
  }

  //-----------------------------------------LEFT-----------
  if (j==0){
    FMUL(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2L, RDST, OPSRCL, f);
    LD(LMADDRLD, RDSTLD, f);
    LMADDRLD = (LMADDRLD + 1) % sizeLMdata;
    RDSTLD   = (RDSTLD + 1) % sizeRFdata;
  }
  //-----------------------------------------RIGHT-----------
  if (j==1){
    FMACCA(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2R, RDST, OPSRCR, f);
    NOP(f);
  }
  //------------------------------------------DOWN---------
  if (j==2){
    FMACCA(RSRC1, 0b01, LMADDR, BROFFSET, BMADDR, RSRC2D, RDST, OPSRCD, f);
    NOP(f);
  }
  //--------------------------------------------UP------
  if (j==3){
    if ((NDST==NDST_UP) || (NDST==NDST_UP_and_RIGHT) || (NDST==NDST_UP_and_LEFT)){
      FMACCA(RSRC1, MODE, LMADDR, BROFFSET, BMADDR, RSRC2U, NDST_UP, OPSRCU, f);
      NOP(f);

    } else if ((NDST==NDST_DOWN) || (NDST==NDST_DOWN_and_RIGHT) || (NDST==NDST_DOWN_and_LEFT)){
      FMACCA(RSRC1, MODE, LMADDR, BROFFSET, BMADDR, RSRC2U, NDST_DOWN, OPSRCU, f);
      NOP(f);
    } else {
      FMACCA(RSRC1, MODE, LMADDR, BROFFSET, BMADDR, RSRC2U, NDST, OPSRCU, f);
      NOP(f);
    }

    if (((i+1)%data_line==0) && ((NDST==NDST_UP_and_RIGHT) || (NDST==NDST_DOWN_and_RIGHT))){
      NOP(f);
      NSG( 0b10, LMADDR-data_line+1, RSRC2, NDST_LEFT, NSRC, f);
      NOP(f);
      NSG( 0b10, LMADDR, RSRC2, NDST_RIGHT, NSRC, f);
    }

    LMADDR = (LMADDR + 1) % sizeLMdata;
  }
  //----------------------------------------------------
  //check if the current chunk of four point is processed
  //as required by the multithreaded computation
  //for populating the four stage accumulation pipeline
  //----------------------------------------------------
  if ((i+1)%4==0)
      {
      if (j!=3)
        {j=j+1;i=i-3;}
      else
        {
        j=0;
        i=i+1;
        }
      }
  else
      {i=i+1;}
  //----------------------------------------------------
}
//----------------------------------------------------
BNZ(f);
NOP(f);
//----------------------------------------------------
NOP(f);
NOP(f);
//----------------------------------------------------
//-----------------------------------------------------
// store data from LM to BM
MODE      = 0b11; //store from LM
LMADDR    = 0;
BMADDR    = 0;
RSRC2     = 0; //not needed here
for (int i=0;i<(data_line*data_line);i++){
  NOP(f);
  STBM( MODE, LMADDR, BMADDR, RSRC2, f);
  LMADDR++;
  BMADDR++;
}
//----------------------------------------------------
REPEAT(16,f);
NOP(f);
//----------------------------------------------------
BNZ(f);
NOP(f);
//----------------------------------------------------
NOP(f);
NOP(f);
//
//----------------------------------------------------
//----------------------------------------------------
//write back results
//32 * 128 beat in total (maximum is 128 beat per burst)
//----------------------------------------------------
// write data from BM to GM
BURSTSIZE = 128;
BMOFFSET  = 0;
GMOFFSETMSB = 0;
GMOFFSETLSB = 0;
for (int i=0;i<32;i++){
  WRGMEM(BURSTSIZE,BMOFFSET,GMOFFSETMSB,f);
  WRGMEM(0,0,GMOFFSETLSB,f);
  //----------------------------------------------------
  ITERATIONS  = WRITE_LATENCY; //cycles to complete transaction
  REPEAT(ITERATIONS,f);
  NOP(f);
  //----------------------------------------------------
  BNZ(f);
  NOP(f);
  //----------------------------------------------------
  NOP(f);
  NOP(f);
  BMOFFSET    = BMOFFSET+32;
  GMOFFSETLSB = GMOFFSETLSB+128*32;
}
//----------------------------------------------------
//the cycles required are finished
//----------------------------------------------------
//----------------------------------------------------
STOP(f);
NOP(f);
//----------------------------------------------------
//    program end
//----------------------------------------------------
f.close();

exit(EXIT_SUCCESS);
}


```

Annex 2: Laplace's equation in two, three and four dimensions

Let us assume we have a given second-order partial differential equation $f(x, y)$ defined as

$$\frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} = 0. \tag{11.1}$$

Eq.11.1 is known as Laplace's equation. To compute a second-order derivative of $f(x, y)$ given on a 3-point stencil (x_{i-1}, x_i, x_{i+1}) following the x-axis, the first derivatives of the first-order accuracy at the intervals (x_{i-1}, x_i) and (x_i, x_{i+1}) are:

$$\left.\frac{\partial f(x,y)}{\partial x}\right|_{x_{i+\frac{1}{2}}y_i} = \frac{f(x_{i+1}, y_i) - f(x_i, y_i)}{\Delta x} \tag{11.2}$$

$$\left.\frac{\partial f(x,y)}{\partial x}\right|_{x_{i-\frac{1}{2}}y_i} = \frac{f(x_i, y_i) - f(x_{i-1}, y_i)}{\Delta x} \tag{11.3}$$

, where Δx is the unit interval. Then, using Eqs. 11.2 and 11.3, the second derivatives can be obtained as

$$\begin{aligned}
\left.\frac{\partial^2 f(x,y)}{\partial x^2}\right|_{x_i y_i} &= \frac{1}{\Delta x}\left\{ \left.\frac{\partial f(x,y)}{\partial x}\right|_{x_{i+\frac{1}{2}}y_i} - \left.\frac{\partial f(x,y)}{\partial x}\right|_{x_{i-\frac{1}{2}}y_i} \right\} \\
&= \frac{1}{\Delta x}\left\{ \frac{f(x_{i+1}, y_i) - f(x_i, y_i)}{\Delta x} - \frac{f(x_i, y_i) - f(x_{i-1}, y_i)}{\Delta x} \right\} \\
&= \frac{1}{\Delta x^2}\{ f(x_{i+1}, y_i) + f(x_{i-1}, y_i) - 2f(x_i, y_i) \}
\end{aligned} \tag{11.4}$$

In the same manner, the second-order derivative of $f(x, y)$ on a 3-point stencil (y_{i-1}, y_i, y_{i+1}) is

$$\left.\frac{\partial^2 f(x,y)}{\partial y^2}\right|_{x_i y_i} = \frac{1}{\Delta y^2}\{ f(x_i, y_{i+1}) + f(x_i, y_{i-1}) - 2f(x_i, y_i) \} \tag{11.5}$$

Here, provided $\Delta x = \Delta y = h$, these Eqs.11.1, 11.4, and 11.5 can derive the second order finite-difference for the Laplacian of 2D functions as

$$\frac{1}{h^2}\{f(x_{i+1}, y_i) + f(x_{i-1}, y_i) + f(x_i, y_{i+1}) + f(x_i, y_{i-1}) - 4f(x_i, y_i)\} = 0 \tag{11.6}$$

With an added fourth dimension z, the 3D Laplace equation can be expressed as:

$$\frac{\partial^2 f(x,y,z)}{\partial x^2} + \frac{\partial^2 f(x,y,z)}{\partial y^2} + \frac{\partial^2 f(x,y,z)}{\partial z^2} = 0. \tag{11.7}$$

Also, with an added fourth dimension w, the 4D Laplace equation becomes:

$$\frac{\partial^2 f(x,y,z,w)}{\partial x^2} + \frac{\partial^2 f(x,y,z,w)}{\partial y^2} + \frac{\partial^2 f(x,y,z,w)}{\partial z^2} + \frac{\partial^2 f(x,y,z,w)}{\partial w^2} = 0. \tag{11.8}$$

By adding the second order derivatives for z and w dimensions, respectively, and provided $\Delta x = \Delta y = \Delta z = \Delta w = h$, it is sufficient to add the terms for the added dimensions and the second order finite-difference for the Laplacian of 3D functions becomes:

$$\frac{1}{h^2}\{f(x_{i+1}, y_i, z_i) + f(x_{i-1}, y_i, z_i) + f(x_i, y_{i+1}, z_i) + f(x_i, y_{i-1}, z_i) \\ + f(x_i, y_i, z_{i+1}) + f(x_i, y_i, z_{i-1}) - 6f(x_i, y_i, z_i)\} = 0 \tag{11.9}$$

and the second order finite-difference for the Laplacian of 4D functions becomes:

$$\frac{1}{h^2}\{f(x_{i+1}, y_i, z_i, w_i) + f(x_{i-1}, y_i, z_i, w_i) + f(x_i, y_{i+1}, z_i, w_i) + f(x_i, y_{i-1}, z_i, w_i) + f(x_i, y_i, z_{i+1}, w_i) + \\ f(x_i, y_i, z_{i-1}, w_i) + f(x_i, y_i, z_i, w_{i+1}) + f(x_i, y_i, z_i, w_{i-1}) - 8f(x_i, y_i, z_i, w_i)\} = 0 \tag{11.10}$$

Bibliography

[1] Heinrich Meyr Tilman Glökler. *Design of Energy-Efficient Application-Specific Instruction Set Processors*. Springer New York, NY, 2004. doi:https://doi.org/10.1007/b105292.

[2] M.J. Irwin. *Low power design for Systems on a Chip*, Sep 1999. Tutorial at the 12th Annual IEEE International ASIC/SOC Conference.

[3] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. A Highly-Efficient and Tightly-Connected Many-Core Overlay Architecture. *IEEE Access*, 9:65277–65292, 2021. doi:10.1109/ACCESS.2021.3074171.

[4] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. A Scalable Many-Core Overlay Architecture on an HBM2-Enabled Multi-Die FPGA. *ACM Trans. Reconfigurable Technol. Syst.*, 16(1), jan 2023. doi:10.1145/3547657.

[5] Xiangwei Li and Douglas Maskell. Time-Multiplexed FPGA Overlay Architectures: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 24:1–19, 07 2019. doi:10.1145/3339861.

[6] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. MITRACA: Manycore Interlinked Torus Reconfigurable Accelerator Architecture. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160-052X, pages 38–38, 2019. doi:10.1109/ASAP.2019.00-35.

[7] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. MITRACA: A Next-Gen Heterogeneous Architecture. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 304–311, 2019. doi:10.1109/MCSoC.2019.00050.

[8] Takahiro Ito, Yoshiki Yamaguchi, Yuetsu Kodama, Junji Yamamoto, Yaoko Nakagawa, Taisuke Boku, and Mitsuhisa Sato. D-6-9 A Study of an ExtremeSIMD Architecture Implemented by an FPGA. In *Proceedings of the IEICE General Conference*, volume 2015, page 73. The Institute of Electronics, Information and Communication Engineers, Feb 2015. URL: `https://ci.nii.ac.jp/naid/110009944858/en/`.

[9] H. M. Waidyasooriya et al. OpenCL-Based FPGA-Platform for Stencil Computation and Its Optimization Methodology. *IEEE Transactions on Parallel and Distributed Systems*, 28(05):1390–1402, May 2017. `doi:10.1109/TPDS.2016.2614981`.

[10] H. M. Waidyasooriya and M. Hariyama. Multi-FPGA Accelerator Architecture for Stencil Computation Exploiting Spacial and Temporal Scalability. *IEEE Access*, 7:53188–53201, Apr 2019. URL: `https://doi.org/10.1109/ACCESS.2019.2910824.Accessedon:Feb26,2021.`, `doi:10.1109/ACCESS.2019.2910824`.

[11] Joseph Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. 01 2005.

[12] T. Sterling, M. Anderson, and M. Brodowicz. *High performance computing: Modern systems and practices*. MorganKaufman., Cambridge, MA, 2018.

[13] James E. Smith. Decoupled Access/Execute Computer Architectures. *ACM TOCS*, 2(4):289–308, November 1984. URL: `https://doi.org/10.1145/357401.357403.Accessedon:Feb26,2021.`, `doi:10.1145/357401.357403`.

[14] Francisco J. Jaime, Javier Hormigo, Julio Villalba, and Emilio L. Zapata. New SIMD instructions set for image processing applications enhancement. In *2008 15th IEEE International Conference on Image Processing*, pages 1396–1399, 2008. `doi:10.1109/ICIP.2008.4712025`.

[15] Samuel Antão and Leonel Sousa. Exploiting SIMD extensions for linear image processing with OpenCL. In *2010 IEEE International Conference on Computer Design*, pages 425–430, 2010. `doi:10.1109/ICCD.2010.5647672`.

[16] Wei Miao, Qingyu Lin, Wancheng Zhang, and Nan-Jian Wu. A Programmable SIMD Vision Chip for Real-Time Vision Applications. *IEEE Journal of Solid-State Circuits*, 43(6):1470–1479, 2008. doi:10.1109/JSSC.2008.923621.

[17] Intel. *Intel Advanced Vector Extensions Programming Reference.* https://www.intel.com/content/dam/develop/external/us/en/documents/36945. Last accessed July 7, 2021.

[18] ARM. *Introducing NEON Development Article.* https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/NEON-architecture-overview/NEON-instructions. Last accessed July 7, 2021.

[19] M. Tremblay, J.M. O'Connor, V. Narayanan, and Liang He. VIS speeds new media processing. *IEEE Micro*, 16(4):10–20, 1996. doi:10.1109/40.526921.

[20] Intel. *Moore's Law and Intel Innovation.* https://www.intel.co.jp/content/www/jp/ja/history/museum-gordon-moore-law.html. Last accessed Sep 29, 2022.

[21] Youngsoo Kim, Shrikant Jadhav, and Clay S. Gloster. Dataflow to Hardware Synthesis Framework on FPGAs. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 91–96, 2016. doi:10.1109/SBAC-PADW.2016.24.

[22] Giuseppe Natale, Marco Bacis, and Marco Domenico Santambrogio. On How to Design Dataflow FPGA-Based Accelerators for Convolutional Neural Networks. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 639–644, 2017. doi:10.1109/ISVLSI.2017.126.

[23] John Teifel and Rajit Manohar. An Asynchronous Dataflow FPGA Architecture. *IEEE Trans. Comput.*, 53(11):1376–1392, nov 2004. doi:10.1109/TC.2004.88.

[24] Chan, Long Chan. Implementing FPGA-optimized Systolic Arrays using 2D Knapsack and Evolutionary Algorithms. Master's thesis, 2022. URL: http://hdl.handle.net/10012/17969.

[25] C.H. Dick. FPGA based systolic array architectures for computing the discrete Fourier transform. In *1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96*, volume 2, pages 465–468 vol.2, 1996. doi:10.1109/ISCAS.1996.541747.

[26] Mahendra Vucha and Arvind Rajawat. Design and FPGA Implementation of Systolic Array Architecture for Matrix Multiplication. *International Journal of Computer Applications*, 26, 07 2011. doi:10.5120/3084-4222.

[27] Martin Langhammer, Sergey Gribok, and Gregg Baeckler. High Density 8-Bit Multiplier Systolic Arrays For Fpga. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 84–92, 2020. doi:10.1109/FCCM48280.2020.00021.

[28] Naohito Nakasato, Hiroshi Daisaka, and Tadashi Ishikawa. High Performance High-Precision Floating-Point Operations on FPGAs Using OpenCL. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 262–265, 2018. doi:10.1109/FPT.2018.00049.

[29] Jiabao Gao, Qingliang Liu, and Jinmei Lai. An Approach of Binary Neural Network Energy-Efficient Implementation. *Electronics*, 10(15), 2021. URL: https://www.mdpi.com/2079-9292/10/15/1830, doi:10.3390/electronics10151830.

[30] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. Low-Precision Floating-Point Arithmetic for High-Performance FPGA-Based CNN Acceleration. *ACM Trans. Reconfigurable Technol. Syst.*, 15(1), nov 2021. doi:10.1145/3474597.

[31] Arif Irwansyah, Vishnu P. Nambiar, and Mohamed Khalil-Hani. An AES Tightly Coupled Hardware Accelerator in an FPGA-based Embedded Processor Core. In *2009 International Conference on Computer Engineering and Technology*, volume 2, pages 521–525, 2009. doi:10.1109/ICCET.2009.248.

[32] Masato Yoshimi, Ryu Kudo, Yasin Oge, Yuta Terada, Hidetsugu Irie, and Tsutomu Yoshinaga. An FPGA-Based Tightly Coupled Accelerator for Data-Intensive Applications. In

2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, pages 289–296, 2014. doi:10.1109/MCSoC.2014.47.

[33] Antoniette Mondigo, Tomohiro Ueno, Kentaro Sano, and Hiroyuki Takizawa. Comparison of Direct and Indirect Networks for High-Performance FPGA Clusters. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th International Symposium, ARC 2020, Toledo, Spain, April 1–3, 2020, Proceedings*, page 314–329, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-44534-8_24.

[34] Marcello Pivanti, F. Schifano, and Hubert Simma. An FPGA-based Torus Communication Network. page 038, 06 2011. doi:10.22323/1.105.0038.

[35] Ahmad Al-Allaf. An FPGA-based Fault Tolerance Hypercube Multiprocessor DSP System. 18:69–83, 01 2010.

[36] Ka-Ming Keung. *A study of on-chip FPGA system with 2D mesh network*. PhD thesis, 2010.

[37] Xilinx. *Alveo U280 Data Center Accelerator Card*, Feb 2021. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html#specifications. Last accessed June 7, 2021.

[38] Ernst Houtgast, Vlad-Mihai Sima, and Zaid Al-Ars. High Performance Streaming Smith-Waterman Implementation with Implicit Synchronization on Intel FPGA using OpenCL. In *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 492–496, 2017. doi:10.1109/BIBE.2017.000-6.

[39] Nikolaos Alachiotis, Simon A. Berger, and Alexandros Stamatakis. Accelerating Phylogeny-Aware Short DNA Read Alignment with FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 226–233, 2011. doi:10.1109/FCCM.2011.13.

[40] Lorenzo Di Tucci, Kenneth O'Brien, Michaela Blott, and Marco D. Santambrogio. Architectural optimizations for high performance and energy efficient Smith-Waterman imple-

mentation on FPGAs using OpenCL. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 716–721, 2017. `doi:10.23919/DATE.2017.7927082`.

[41] Riadh Ben Abdelhamid and Yoshiki Yamaguchi. A Block-Based Systolic Array on an HBM2 FPGA for DNA Sequence Alignment. In Fernando Rincón, Jesús Barba, Hayden K. H. So, Pedro Diniz, and Julián Caba, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 298–313, Cham, 2020. Springer International Publishing.

[42] Mohammed Alser, Taha Shahroodi, Juan Gómez-Luna, Can Alkan, and Onur Mutlu. SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs. *CoRR*, abs/1910.09020, 2019. URL: `http://arxiv.org/abs/1910.09020`, `arXiv:1910.09020`.

[43] Peng Lei, Jiawei Liang, Zhenyu Guan, Jun Wang, and Tong Zheng. Acceleration of FPGA Based Convolutional Neural Network for Human Activity Classification Using Millimeter-Wave Radar. *IEEE Access*, 7:88917–88926, 2019. `doi:10.1109/ACCESS.2019.2926381`.

[44] Wenjin Huang, Huangtao Wu, Qingkun Chen, Conghui Luo, Shihao Zeng, Tianrui Li, and Yihua Huang. FPGA-Based High-Throughput CNN Hardware Accelerator With High Computing Resource Utilization Ratio. *IEEE Transactions on Neural Networks and Learning Systems*, 33(8):4069–4083, 2022. `doi:10.1109/TNNLS.2021.3055814`.

[45] Mengshu Sun, Pu Zhao, Mehmet Gungor, Massoud Pedram, Miriam E. Leeser, and X. Lin. 3D CNN Acceleration on FPGA using Hardware-Aware Pruning. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[46] Hongxiang Fan, Xinyu Niu, Qiang Liu, and Wayne Luk. F-C3D: FPGA-based 3-dimensional convolutional neural network. *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.

[47] S.C. Chan, H.O. Ngai, and K.L. Ho. A programmable image processing system using FPGA. In *Proceedings of IEEE International Symposium on Circuits and Systems - ISCAS '94*, volume 2, pages 125–128 vol.2, 1994. `doi:10.1109/ISCAS.1994.408921`.

[48] Mohammad I. AlAli, Khaldoon M. Mhaidat, and Inad A. Aljarrah. Implementing image processing algorithms in FPGA hardware. In *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–5, 2013. `doi:10.1109/AEECT.2013.6716446`.

[49] Hayato Hagiwara, Kenichi Asami, and Mochimitsu Komori. Real-time image processing system by using FPGA for service robots. In *The 1st IEEE Global Conference on Consumer Electronics 2012*, pages 720–723, 2012. `doi:10.1109/GCCE.2012.6379964`.

[50] Faraz Bhatti and Thomas Greiner. FPGA Hardware Design for Plenoptic 3D Image Processing Algorithm Targeting a Mobile Application. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7863–7867, 2021. `doi:10.1109/ICASSP39728.2021.9414690`.

[51] Kentaro Sano et al. Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):695–705, 2014. `doi:10.1109/TPDS.2013.51`.

[52] Kentaro Sano et al. FPGA-Array with Bandwidth-Reduction Mechanism for Scalable and Power-Efficient Numerical Simulations Based on Finite Difference Methods. *ACM TRETS*, 3(4), Nov 2010. `doi:10.1145/1862648.1862651`.

[53] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. When FPGA Meets Cloud: A First Look at Performance. *IEEE Transactions on Cloud Computing*, 10(2):1344–1357, 2022. `doi:10.1109/TCC.2020.2992548`.

[54] Zhuangdi Zhu, Alex X. Liu, Fan Zhang, and Fei Chen. FPGA Resource Pooling in Cloud Computing. *IEEE Transactions on Cloud Computing*, 9(2):610–626, 2021. `doi:10.1109/TCC.2018.2874011`.

[55] Abid Farhan, Raafat Aburukba, Assim Sagahyroon, Mohammed Elnawawy, and Khaled El-Fakih. Virtualizing and Scheduling FPGA Resources in Cloud Computing Datacenters. *IEEE Access*, 10:96909–96929, 2022. `doi:10.1109/ACCESS.2022.3204866`.

[56] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116, 2014. `doi:10.1109/FCCM.2014.42`.

[57] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435, 2015. `doi:10.1109/CloudCom.2015.60`.

[58] Guohao Dai, Yi Shan, Fei Chen, Yu Wang, Kun Wang, and Huazhong Yang. Online scheduling for FPGA computation in the Cloud. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 330–333, 2014. `doi:10.1109/FPT.2014.7082811`.

[59] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Trans. Reconfigurable Technol. Syst.*, 15(3), feb 2022. `doi:10.1145/3506713`.

[60] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1078–1086, 2015. `doi:10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199`.

[61] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. Network-attached FPGAs for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 36–43, 2016. `doi:10.1109/FPT.2016.7929186`.

[62] Xilinx. *7 Series DSP48E1 Slice User Guide*. `https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf`. Last accessed Sep 20, 2022.

[63] Xilinx. *UltraScale Architecture DSP Slice User Guide.* `https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf`. Last accessed Sep 20, 2022.

[64] Xilinx. *UltraScale Architecture Configurable Logic Block User Guide*, Feb 2017. `https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf`. Last accessed Dec 16, 2019.

[65] Xilinx. *UltraScale Architecture Memory Resources.* `https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf`. Last accessed Dec 16, 2019.

[66] Xilinx. *UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices*, June 2016. `https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf`. Last accessed 16 Dec 2019.

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

[68] Abhishek Kumar Jain et al. Throughput oriented FPGA overlays using DSP blocks. In *2016 Design, Automation Test in Europe Conference Exhibition*, pages 1628–1633. IEEE, 2016.

[69] Kumar H B Chethan and Nachiket Kapre. Hoplite-DSP: Harnessing the Xilinx DSP48 multiplexers to efficiently support NoCs on FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, 2016. `doi:10.1109/FPL.2016.7577317`.

[70] J. Bachrach et al. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, June 2012. `doi:10.1145/2228360.2228584`.

[71] Kavya et al Shagrithaya. Enabling development of OpenCL applications on FPGA platforms. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 26–30, 2013. `doi:10.1109/ASAP.2013.6567546`.

[72] K. Paul, C. Dash, and M. S. Moghaddam. reMORPH: A Runtime Reconfigurable Architecture. In *2012 15th Euromicro Conference on Digital System Design*, pages 26–33, Sep. 2012. `doi:10.1109/DSD.2012.111`.

[73] Kalin Ovtcharov, Ilian Tili, and J. Gregory Steffan. TILT: A multithreaded VLIW soft processor family. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, 2013. `doi:10.1109/FPL.2013.6645553`.

[74] Charles Laforest and Jason Anderson. Microarchitectural Comparison of the MXP and Octavo Soft-Processor FPGA Overlays. *ACM Transactions on Reconfigurable Technology and Systems*, 10:1–25, 05 2017. `doi:10.1145/3053679`.

[75] Jan Gray. GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator. In *Proc. FCCM*, pages 17–20, Washington, DC, USA, May 2016. `doi:10.1109/FCCM.2016.12`.

[76] Jan Gray. *2GRVI Phalanx: A 1332-Core RISC-V RV64I Processor Cluster Array with an HBM2 High Bandwidth Memory System, and an OpenCL-like Programming Model, in a Xilinx VU37P FPGA [WIP Report]*. Denver, CO, USA, Nov 2019.

[77] Sunil Shukla, Neil Bergmann, and Juergen Becker. QUKU: A Coarse Grained Paradigm for FPGAs. 01 2006.

[78] M. A. Kinsy, M. Pellauer, and S. Devadas. Heracles: Fully Synthesizable Parameterized MIPS-Based Multicore System. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 356–362, Sep. 2011. `doi:10.1109/FPL.2011.70`.

[79] James Coole and Greg Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 13–22, 01 2010. `doi:10.1145/1878961.1878966`.

[80] Alexander Brant. *Coarse and fine grain programmable overlay architectures for FPGAs*, 2013. https://dx.doi.org/10.14288/1.0073573.

[81] Davor Capalija and Tarek Abdelrahman. Towards Synthesis-Free JIT Compilation to Commodity FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 202 – 205, 06 2011. doi:10.1109/FCCM.2011.25.

[82] Alexander Brant and Guy Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 93–96, 04 2012. doi:10.1109/FCCM.2012.25.

[83] Chethan Kumar H B, Prashant Ravi, Gourav Modi, and Nachiket Kapre. 120-Core MicroAptiv MIPS Overlay for the Terasic DE5-NET FPGA Board. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 141–146, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3020078.3021751.

[84] Jan Gray. *2GRVI Phalanx: W.I.P Towards kilocore RISC-V FPGA Accelerators with HBM2 DRAM.* http://fpga.org/wp-content/uploads/2019/08/HotChips31-2GRVI-Phalanx-poster.pdf. Last accessed Dec 16, 2019.

[85] Jun Makino, Kei Hiraki, and Mary Inaba. GRAPE-DR: 2-Pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007. doi:10.1145/1362622.1362647.

[86] *A feasibility study for future HPCI system with an extreme accelerator, A working group for future HPCI system promotion*, Mar 2013. https://www.mext.go.jp/b_menu/shingi/chousa/shinkou/028/shiryo/1332667.htm. Last accessed Dec 27, 2022.

[87] Yuetsu Kodama, Yoshiki Yamaguchi, Naohito Nakasato, Junichiro Makino, Taisuke Boku, and Mitsuhisa Sato. *A Study of Extreme SIMD Accelerator*, August 2013. Japanese.

[88] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. Condensing an overload of parallel computing ingredients into a single architecture recipe. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 25–28, 2020. `doi:10.1109/ASAP49362.2020.00013`.

[89] Paul Havinga and Gerard Smit. Design techniques for low-power systems. *Journal of Systems Architecture*, 46:1–21, 08 2000. `doi:10.1016/S1383-7621(98)00057-5`.

[90] Intel. *Cyclone V SoC Power Optimization*, Sep 2015. `https://www.intel.com/content/www/us/en/docs/programmable/683713/current/fpga-power-consumption.html`. Last accessed Feb 8, 2023.

[91] Arash Farhadi Beldachi and Jose L. Nunez-Yanez. Run-time power and performance scaling in 28 nm FPGAs. *IET Computers & Digital Techniques*, 8(4):178–186, 2014. `doi:https://doi.org/10.1049/iet-cdt.2013.0117`.

[92] Ivan Ratković, Nikola Bezanic, Osman S. Unsal, Adrián Cristal, and Veljko M. Milutinovic. Chapter One - An Overview of Architecture-Level Power- and Energy-Efficient Design Techniques. *Adv. Comput.*, 98:1–57, 2015.

[93] R.I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 218–229, 2001. `doi:10.1109/ISCA.2001.937451`.

[94] Hai Li, S. Bhunia, Y. Chen, T.N. Vijaykumar, and K. Roy. Deterministic clock gating for microprocessor power reduction. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 113–122, 2003. `doi:10.1109/HPCA.2003.1183529`.

[95] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pages 132–141, 1998. `doi:10.1109/ISCA.1998.694769`.

[96] J.L. Aragon, J. Gonzalez, and A. Gonzalez. Power-aware control speculation through selective throttling. In *The Ninth International Symposium on High-Performance Computer*

Architecture, 2003. HPCA-9 2003. Proceedings., pages 103–112, 2003. `doi:10.1109/HPCA.2003.1183528`.

[97] Zhigang Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (IEEE Cat. No.04TH8758)*, pages 32–37, 2004. `doi:10.1145/1013235.1013249`.

[98] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors. In *Proceedings of the 11th Great Lakes Symposium on VLSI*, GLSVLSI '01, page 73–78, New York, NY, USA, 2001. Association for Computing Machinery. `doi:10.1145/368122.368807`.

[99] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 90–101, 2001. `doi:10.1109/MICRO.2001.991108`.

[100] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 230–239, 2001. `doi:10.1109/ISCA.2001.937452`.

[101] Jungwook Kim, Seong Tae Jhang, and Chu Shik Jhon. Dynamic Register-Renaming Scheme for Reducing Power-Density and Temperature. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, page 231–237, New York, NY, USA, 2010. Association for Computing Machinery. `doi:10.1145/1774088.1774134`.

[102] Naresh Grover and M. K.Soni. Reduction of Power Consumption in FPGAs - An Overview. *International Journal of Information Engineering and Electronic Business*, 4:50–69, 10 2012. `doi:10.5815/ijieeb.2012.05.07`.

[103] P. Biswas, S. Banerjee, N. Dutt, P. Ienne, and L. Pozzi. Performance and energy benefits of instruction set extensions in an FPGA soft core. In *19th International Conference on*

VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06), pages 6 pp.–, 2006. doi:10.1109/VLSID.2006.131.

[104] Robert G. Dimond, Oskar Mencer, and Wayne Luk. Combining Instruction Coding and Scheduling to Optimize Energy in System-on-FPGA. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 175–184, 2006. doi:10.1109/FCCM.2006.31.

[105] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture.* PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html. Last accessed 16 Dec 2019.

[106] Nvidia. *NVIDIA TESLA V100 GPU ARCHITECTURE*, Aug 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Last accessed Dec 3, 2022.

[107] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sep. 1972. doi:10.1109/TC.1972.5009071.

[108] Lei Cheng, Deming Chen, and Martin D.F. Wong. GlitchMap: An FPGA Technology Mapper for Low Power Considering Glitches. In *2007 44th ACM/IEEE Design Automation Conference*, pages 318–323, 2007.

[109] Rollins, Nathaniel Hatley. Reducing power in FPGA designs through glitch reduction. Master's thesis, 2007. URL: https://scholarsarchive.byu.edu/etd/1105.

[110] Shen, Ghosh, Devadas, and Keutzer. On average power dissipation and random pattern testability of CMOS combinational logic networks. In *1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 402–407, 1992. doi:10.1109/ICCAD.1992.279338.

[111] Xilinx. *UltraFast Design Methodology Guide for the Vivado Design Suite.* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug949-vivado-design-methodology.pdf. Last accessed Dec 16, 2019.

[112] Xilinx. *Vitis High-Level Synthesis User Guide*, Dec 2022. `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Interfaces-for-Vitis-Kernel-Flow`. Last accessed Dec 27, 2022.

[113] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, Feb 1990. `doi:10.1109/2.44900`.

[114] Zhipeng Gong, Tefang Chen, Fumin Zou, Li Li, and Yingxi Kang. Implementation of Multi-channel FIFO in One BlockRAM with Parallel Access to One Port. *Journal of Computers*, 9, 05 2014. `doi:10.4304/jcp.9.5.1193-1200`.

[115] Xilinx. *UltraScale Architecture Memory Resources*, February 2021. `https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf`. Last accessed June 20, 2021.

[116] AXIprotocol. *AMBA AXI and ACE Protocol Specification*, 2013. `https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process?lang=en`.

[117] Mustafa Abbas and Vaughn Betz. Latency Insensitive Design Styles for FPGAs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 360–3607, New York, NY, USA, 2018. IEEE. `doi:10.1109/FPL.2018.00068`.

[118] Changdao Du et al. FPGA-Based Computational Fluid Dynamics Simulation Architecture via High-Level Synthesis Design Method. In Fernando Rincón et al., editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 232–246. Springer International Publishing, 2020.

[119] Antoniette Mondigo et al. Scalability Analysis of Deeply Pipelined Tsunami Simulation with Multiple FPGAs. *IEICE Transactions on Information and Systems*, E102.D:1029–1036, 05 2019. `doi:10.1587/transinf.2018RCP0007`.

[120] Johannes Pekkilä et al. *Scalable communication for high-order stencil computations using CUDA-aware MPI*, 2021. `arXiv:2103.01597`.

[121] T. Sterling et al. *High performance computing: Modern systems and practices*, pages 294–295. Morgan Kaufmann, 2018.

[122] Zeke Wang et al. Shuhai: Benchmarking High Bandwidth Memory On FPGAS. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 111–119. IEEE, 2020. `doi:10.1109/FCCM48280.2020.00024`.

[123] Mariem Saied. *Automatic code generation and optimization of multi-dimensional stencil computations on distributed-memory architectures.* PhD thesis, University of Strasbourg, Strasbourg, France, 2018.

[124] Jeremy Fowers et al. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, page 47–56. ACM, 2012. `doi:10.1145/2145694.2145704`.

[125] Justin Holewinski et al. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *ACM International Conference on Supercomputing*, page 311–320. ACM, 2012. `doi:10.1145/2304576.2304619`.

[126] Riccardo Cattaneo et al. On How to Accelerate Iterative Stencil Loops: A Scalable Streaming-Based Approach. *ACM Trans. Archit. Code Optim.*, 12(4), dec 2015. `doi:10.1145/2842615`.

[127] Yuze Chi et al. SODA: Stencil with Optimized Dataflow Architecture. In *International Conference on Computer-Aided Design*, ICCAD '18, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3240765.3240850`.

[128] Enrico Reggiani et al. Enhancing the Scalability of Multi-FPGA Stencil Computations via Highly Optimized HDL Components. *ACM Trans. Reconfigurable Technol. Syst.*, 14(3), aug 2021. `doi:10.1145/3461478`.

[129] Hamid Reza Zohouri et al. Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL. In *ACM/SIGDA International Symposium*

on Field-Programmable Gate Arrays, page 153–162. ACM, 2018. doi:10.1145/3174243.3174248.

[130] Intel. *APP Metrics for Intel Microprocessors Intel Core Processor*, 2021. https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf. Last accessed December 23, 2021.

[131] Intel. *Intel Arria 10 Product Table*, 2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf. Last accessed December 23, 2021.

[132] Intel. *APP Metrics for Intel Microprocessors Intel Xeon Processor*, 2021. https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Xeon-Processors.pdf. Last accessed December 23, 2021.

[133] Nvidia. *NVIDIA Tesla P100 GPU ACCELERATOR*, Oct 2016. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf. Last accessed Dec 23, 2021.

[134] Nvidia. *NVIDIA V100 TENSOR CORE GPU*, Jan 2020. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf. Last accessed Dec 23, 2021.

[135] Intel. *Stratix V Device Overview*, 2020. https://www.mouser.com/datasheet/2/612/stx5_51001-1099064.pdf. Last accessed December 23, 2021.

[136] Gokhan Apaydin and Levent Sevgi. *Wave Propagation Inside Three-Dimensional Rectangular Waveguide*, pages 79–88. 2018. doi:10.1002/9781119432166.ch7.

[137] Paul Adrien Maurice Dirac. Relativistic wave equations. *Proceedings of the Royal Society of London. Series A - Mathematical and Physical Sciences*, 155(886):447–459, 1936. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1936.0111, arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.1936.0111, doi:10.1098/rspa.1936.0111.

[138] Gowri Shankar Ramaswamy and F. Sagayaraj Francis. The idea of spacetime in conceptual knowledge. In *2014 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–4, 2014. `doi:10.1109/ICCIC.2014.7238354`.