Technical University of Crete

Diploma Thesis

Main Memory Performance for Realistic Data Access in FPGA Systems: An Experimental Study

Author:
Maria Argyriou

Thesis Committee:
Prof. Apostolos Dollas
Prof. Michalis Zervakis
Assoc. Prof. Sotirios Ioannidis

A thesis submitted in fulfillment of the requirements for the DIPLOMA of Electrical and Computer Engineering in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Lab

April 14, 2021

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

DIPLOMA THESIS

Main Memory Performance for Realistic Data Access in FPGA Systems: An Experimental Study

by Maria Argyriou

Nowadays, computationally demanding applications, such as Convolutional Neural Networks (CNN), are mapped to hardware accelerators like Field Programmable Gate Arrays (FPGAs) due to customizable datapath with designer-tunable parallelism and pipelining. Memory is in many cases the limiting factor of every performance-bound application but its real performance is often overlooked. Most studies and benchmarks on memory subsystems focus on best-case scenarios for memory access times. Memory access times and throughput are affected by such factors as the memory controller's performance, buffering, the need (or lack of) a microprocessor for on-FPGA data transfer, and even the capabilities of Computer-Aided Design (CAD) tools and frameworks. This study, prompted by CNN applications on mid-range single- and multi-FPGA systems focuses on the experimental memory evaluation, aiming at providing the designer with realistic figures which can be used towards on-FPGA buffer sizing, computation-to-memory I/O estimation to avoid bottlenecks, and even pipeline strategy. Detailed experimental results have been obtained by memory access patterns which represent realistic scenarios, and these are presented and analyzed in this thesis. One of the conclusions from this work is that when random accesses are required, large numbers of such accesses performed together lead to better results vs. fewer, scattered accesses. The results of this work not only show a significant deviation from ideal transfer rates of the DDR memory channel (which can approach 20 GBytes/sec), but they are also substantially lower than the internal AXI port maximum bandwidth of 4.8 GBytes/sec, the degradation being due to internal to the FPGA data transfers and the DDR controller.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

DIPLOMA THESIS

Main Memory Performance for Realistic Data Access in FPGA Systems: An Experimental Study

by Maria Argyriou

Σήμερα, επιταχυντές υλικού όπως οι Field Programmable Gate Arrays (FPGAs) χρησιμοποιούνται σε υπολογιστικά απαιτητικές εφαρμογές, όπως τα Convolutional Neural Networks (CNN), εξαιτίας της προσαρμοστικότητάς τους και των δυνατοτήτων που προσφέρουν στον σχεδιαστή για παραλληλισμό και pipelining. Η μνήμη είναι σε πολλές περιπτώσεις ο περιοριστικός παράγοντας για εφαρμογές όπου η ταχύτητα των επιταχυντών είναι σημαντική, αλλά η πραγματική απόδοσή της συχνά παραβλέπεται. Οι περισσότερες έρευνες σε υποσυστήματα μνήμης επικεντρώνονται σε μελέτες όπου οι χρόνοι πρόσβασης στη μνήμη είναι βέλτιστοι. Οι χρόνοι αυτοί και η απόδοσή της επηρεάζονται από παράγοντες όπως ο controller της μνήμης, το buffering, η ανάγκη (ή η έλλειψη) ενός μικροεπεξεργαστή για on-FPGA μεταφορά δεδομένων και από τις δυνατότητες των εργαλείων σχεδίασης. Η παρούσα μελέτη, παρακινούμενη από εφαρμογές που υλοποιούν CNN αρχιτεκτονικές σε συστήματα με μια ή περισσότερες FPGA, επικεντρώνεται στην πειραματική αξιολόγηση της μνήμης, με στόχο να παρέχει στον σχεδιαστή ρεαλιστικά αποτελέσματα που μπορούν να χρησιμοποιηθούν για την εύρεση του μεγέθους των buffers στις FPGA, την αποφυγή bottlenecks μέσα από τον υπολογισμό του I/O, ακόμη και την επιλογή της κατάλληλης pipelining μεθοδολογίας. Τα πειραματικά αποτελέσματα προσομοιώνουν μοτίβα πρόσβασης της μνήμης που αντιπροσωπεύουν ρεαλιστικά σενάρια, τα οποία παρουσιάζονται και αναλύονται σε αυτήν την εργασία. Ένα από τα συμπεράσματα από αυτήν την εργασία είναι ότι όταν απαιτούνται τυχαίες προσβάσεις, μεγάλος αριθμός τέτοιων προσπελάσεων που εκτελούνται μαζί οδηγούν σε καλύτερα αποτελέσματα έναντι λιγότερων διάσπαρτων προσβάσεων. Τα αποτελέσματα αυτής της εργασίας όχι μόνο δείχνουν αξιοσημείωτη απόκλιση από τους ιδανικούς ρυθμούς μεταφοράς δεδομένων από τη μνήμη (οι οποίοι μπορεί να προσεγγίσουν τα 20 GBytes/sec), αλλά είναι επίσης σημαντικά χαμηλότερα από το μέγιστο bandwidth της AXI θύρας (4,8 GBytes/sec). Η υποβάθμιση αυτή οφείλεται στον τρόπο μεταφοράς δεδομένων στις FPGA και στον controller της μνήμης.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Apostolos Dollas, for his continuous support and insightful guidance throughout this thesis and the course of my studies in the ECE department. I would also like to express my gratitude since his knowledge and experiences were one of the reasons I decided to work in Hardware Architecture design.

Furthermore, I would like to thank the committee, Prof. Michalis Zervakis, and Assoc. Prof. Sotirios Ioannidis, for evaluating the work of this thesis.

In addition, I would like to express my thankfulness to the CARV team in FORTH for their guidance throughout this thesis, especially Dr. Aggelos Ioannou, for his support and expertise in the field of hardware design, Dr. Christos Kozanitis and, Dr. Gregory Tsagkatakis for their support and insightful inputs.

This work could not have been completed without the TUC MHL laboratory's valuable support, especially Ph.D. candidates Pavlos Malagonakis and Andreas Brokalakis, whose help in understanding and using Xilinx's tools and the ZEUS server was invaluable.

I would also like to thank my fellow colleagues of the TUC MHL laboratory, Charisios Loukas and Tzanis Fotakis, for their help and our excellent collaboration.

Last but not least, I would like to express my deepest gratitude to my family, especially my parents and my brother, for their continuous support throughout the years and my friends for their patience and understanding.

Maria Argyriou,
Chania 2021

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
APU	**A**pplication **P**rocessing **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced e**X**tensible **I**nterface
BRAM	**B**lock **R**andom **A**ccess **M**emory
CAS	**C**olumn **A**ddress **S**trobe
CCL	**C**onnected **C**omponent **L**abeling
CPU	**C**entral **P**rocessor **U**nit
DDR4	**D**ouble **D**ata **R**ate type **4** memory
DMA	**D**irect **M**emory **A**ccess
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
FPD	**F**ull **P**ower **D**omain
FPGA	**F**ield **P**rogrammable **L**ogic **D**esign
FSM	**F**inite **S**tate **M**achine
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**igh **P**erformance **C**omputing
HRPC	**H**igh **P**erformance **R**econfigurable **C**omputer
ILA	**I**ntegrated **L**ogic **A**nalyzer
IoT	**I**nternet **o**f **T**hings
LPD	**L**ow **P**ower **D**omain
MPSoC	**M**ulti **P**rocessor **S**ystem on **C**hip
PE	**P**rocessing **E**lement
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
QFDB	**Q**uad **F**PGA **D**aughter **B**oard
RAS	**R**ow **A**ddress **S**trobe
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SDRAM	**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory

Dedicated to Carol, whom I will always miss...

Chapter 1

Introduction

In the last decade, the amount of data produced is increased every single day. According to researches, the entire digital universe is expected to reach 46 zettabytes by 2021, making the need for accurate handling vital. In a span of two years (2016-2018), 90% of the world's data was created [1]. For example, in 2018, Google processed 3.5 billion searches every day. It is estimated that by the end of 2020, more than 20 billion devices will utilize the IoT producing more data than before. Other computationally demanding areas such as high-speed search, machine learning, and AI; high-performance computing in data centers; real-time graphics processing, including virtual reality and video gaming; and, soon, autonomous vehicles are starting or already producing vast amounts of information. Managing this amount of data seems almost impossible without the use of state of the art computers and computation methods.

The technological progress that has been made in recent years enables the processing of a vast amount of information through supercomputer systems and algorithms. A typical example is neural networks, the usefulness of which covers a wide range of applications, from the classification of geographical areas through photographs[1] to the simulation of complex board games[2]. The design of an application-specific accelerator circuit may be arduous and expensive, but offer better performance or power consumption vs. CPUs or GPUs. Application Specific Integrated Circuit (ASIC) is designed for a specific purpose; it cannot be re-programmed and function the same for their whole operating life. FPGAs are integrated circuits that can be configured to solve a computational problem. CPUs and GPUs are instruction-based hardware configured via software, whereas FPGAs are configured by specifying a hardware circuit. This reconfigurability makes FPGAs an excellent choice for applications in which standards are evolving, such as digital television, consumer electronics, cyber security systems, and wireless communications. Although the systems we have today demonstrate excellent progress,

[1] https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=6ee88c2760ba

many technological problems remain unresolved, or their solution is yet to be optimized. A typical example is that an FPGA can implement a high-speed neural network, but the rate of data in which they are fed to the network is much slower than the processing rate meaning that the volume of data and its routing through the network remains a problem.

1.1 Motivation and Scientific Contribution

Memory is one of the most significant performance limiting factors of FPGA-based designs. During our research, we realized that the memory subsystem of modern-day FPGAs is often overlooked; researchers are focused on implementing fast and efficient hardware accelerators to produce results by neglecting the bottlenecks caused due to memory bandwidth limitations. Our goal is to evaluate the memory subsystem of a modern and often used FPGA, provide the necessary information concerning the most efficient memory utilization, and test various techniques using Memory Mapped IPs that simulate FPGA memory resources. This work was prompted by actual CNN applications which were under development at ICS/FORTH and the Technical University of Crete.

This thesis has experimental work which was performed in order to quantify memory subsystem performance in two platforms, the widely-used Xilinx ZCU-102[3] and the proprietary EuroEXA-project developed QFDB[4]. Both use a very common mid-size FPGA, whereas the CAD tools that we used are most typical of the entire gamut of Xilinx FPGAs. It was mandated by the need to schedule accesses and size BRAM buffers in signal classification with CNNs [5] applications, however, there are many more applications that require multi-dimensional memory access patterns, in which only one of the dimensions can be conveniently mapped to the DDR row and hence benefit from burst transfers. This work's contribution is the experimental quantification of memory subsystem performance on a mid-scale Xilinx FPGA coupled to a corresponding DDR main memory and with a very recent version of the Vivado[6] CAD tool suite. This evaluation accounts for all internal and external interfaces, as well as the effect of CAD tools in the design. Thus, it can be used towards on-FPGA BRAM buffer sizing, computation-to-memory I/O ratio to avoid bottlenecks, and pipeline strategy.

The goal of this thesis is to evaluate the memory subsystem of an FPGA that is used by architects to deploy various computationally demanding architectures. For this purpose we performed:

- Timing tests and evaluation of the on-chip memory module and controller of a specific FPGA architecture

- Experiments using different memory access methods and their trade-offs, from sequential to random memory access using various techniques
- Ways to transfer data from the main memory to the on-chip memory that resides inside the FPGA, maintaining low energy consumption and the highest achievable memory bandwidth
- Modelization of the effect of access patterns on memory performance and provided and explanation on how it can be exploited by the designers

Our experiments revealed how the memory controller interacts with the main memory module and the memory that resides in the programmable logic part of an FPGA. Furthermore, we gained hardware expertise since we deduced information on how the memory operates, how memory is affected by different types of memory access and how the controller receives and handles these requests. What is more, details on how to improve PS memory utilization and mapping are presented along with efficient ways to use the memory controller, helping future hardware developers.

1.2 Thesis Outline

Below we present the thesis outline, per chapter:

- **Chapter 2 - Theoretical Background:** In this chapter, the theoretical background of Memory components and organization in an FPGA is described.
- **Chapter 3 - Related Work:** Related work and scientific contributions are described.
- **Chapter 4 - FPGA Implementation:** In this chapter we explain the tools we used in our experiments.
- **Chapter 5 -Memory Subsystem Evaluation:** The memory subsystem is evaluated, from sequential to random access using various memory attributes and techniques.
- **Chapter 6 - Results:** Metrics, such as throughput, latency, and power consumption, are presented alongside the results of each type of memory access. Sequential and random (using memory strides) access is evaluated, simulating computationally demanding small and large CNNs.
- **Chapter 7 - Conclusions and Future Work:** This thesis is being concluded and we present directions and ideas for future work or possible extensions.

Chapter 2

Theoretical Background

This chapter describes the theoretical background of memory architecture, the types of memory that exist nowadays, the memory components, and their organization in the current FPGA's.

2.1 Memory Organization in Computer Architecture

ENIAC was the first programmable digital computer that could perform simple calculations using 20 numbers with ten decimal digits, which were held in the vacuum tube accumulators.

FIGURE 2.1: Detail of the back of a section of ENIAC, showing vacuum tubes:URL

One of the essential parts of a computer is the memory component. Computer memory is a physical device that can store information either temporarily or permanently. Volatile memory loses its contents when the hardware device is switched off, whereas non-volatile memory (which was introduced in the late 40s - early 50s and quickly became the dominant form of memory) keeps its contents

even if the power is lost. Since CPU must operate at maximum speed for the highest performance, uninterrupted high-speed access is required.

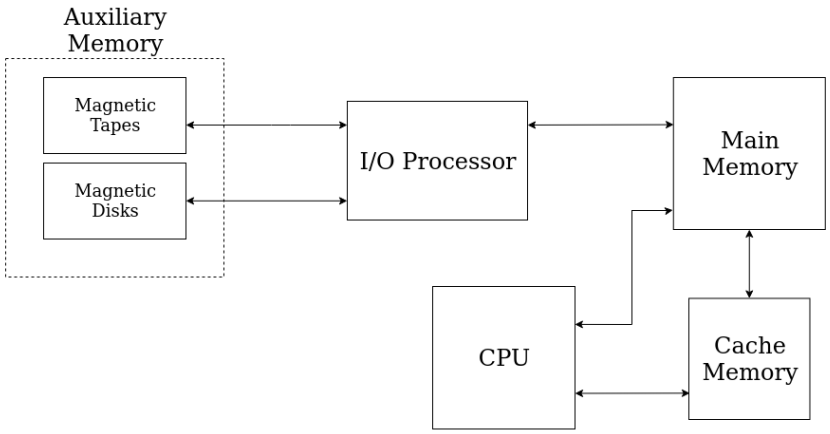


FIGURE 2.2: Memory communication with CPU.

2.1.1 Secondary Storage (Auxiliary Memory)

This memory component is at the bottom of the hierarchy, it is not directly connected to the CPU, and it trades slower access rates (its access time is 1000 times the access time of the main memory) for higher storage capacity and data stability. Since it is a non-volatile memory type, it holds data that are not needed by the main memory for future use. It can either be accessed sequentially or directly. The hard disk drive is a type of auxiliary memory.

2.1.2 Main Memory

Main or central memory is directly connected to the CPU, auxiliary memory, and cache. It is used to store software applications and other information for the CPU to have fast and direct access when needed. The main memory consists of RAM and ROM.

2.1.2.1 Random Access Memory

Data items are read and written in the same amount of time regardless of the physical location of the data. RAM devices have a set of address lines, and a set of memory cells is activated for each combination of bits (due to this RAM chip has a capacity that is a power of two). Each cell has a unique address that can be found by counting across columns and then counting down by row. This memory type is volatile and can be divided into dynamic RAM (DRAM) and static RAM (SRAM).

- SRAM is more expensive, faster to access, and requires more space (meaning that for the same amount of transistors, DRAM will have 4x - 6x the capacity of SRAM) for a given amount of data than DRAM. It is used for L1 and L2 caches, and the typical access time is 10 ns. SRAM modules consume less power than DRAM modules since they require small and steady current. Power consumption increases when SRAM is operated at higher frequencies.
- On the other hand, the controller (either CPU or memory) must recharge the capacitors before they discharge in order to maintain the data in a DRAM chip, meaning that the memory controller reads and rewrites the data. This refreshing technique occurs thousands of times per second, causing the DRAM chip to be slower than SRAM. The average access time is about 60 ns.

2.1.2.2 Read-Only Memory

This type of memory is non-volatile, and its contents have been prerecorded. ROM is used to store the instruction required during bootstrap of the computer, and its speed is much slower than RAM. CPU is not directly connected with ROM, and in order to access its data, they must be transferred to the RAM component. ROM has three types, PROM (Programmable read-only memory), EPROM (Erasable Programmable read-only memory) and EEPROM (Electrically erasable programmable read-only memory).

- PROM can be programmed by the user and data cannot be changed.
- EPROM can be reprogrammed and in order to erase data from it, the chip must be exposed to ultra violet light.
- EEPROM can be reprogrammed and in order to do so, electric field must be applied (only portions of the chip are erased).

2.1.2.3 Memory Access Methods

A hardware function's performance and the data transfer rate between PS and PL depend on accessing the memory efficiently. There are four types of memory access methods, as presented below.

- **Sequential Access:** In sequential access, the data are retrieved in a specific linear manner (like accessing data stored in a single Linked List), and their location affects access time.
- **Random Access:** This method provides the ability to access any arbitrary element of a sequence at the same access time no matter how many elements may be in the set.

- **Direct Access:** Data retrieved with direct access can be obtained by directly referring to where they are physically located on the memory. Direct access is a combination of sequential and random access, and the access time depends on the characteristics of the storage technology and the memory organization.
- **Associate Access:** In this memory access type, the stored data can be identified for access by the content of the data itself rather than by an address or memory location.

We experimented with sequential and random data access in ZCU102, measuring the access time for small and big chunks of data retrieved from the DDR. The results are presented in Chapter 6.

2.1.3 Cache Memory

Cache is a high-speed memory component, and it is either built into a computer's CPU or located next to it on a separate chip. Cache memory is used to store instructions that are repeatedly required to run programs, improving overall speed and the performance of multi-core systems. It is not as costly as registers but faster when compared to main memory. This means that the CPU does not have to use the bus on the motherboard for data transfer, because whenever data must be passed through the system bus, the data transfer speed slows the motherboard's capability. In computer architecture, Level 1 (L1) cache is built into the CPU, and Level 2 (L2) resides on a separate chip next to the CPU. Some CPUs have both L1 and L2 cache built-in and designate separate cache chip as Level 3 (L3). Since built-in cache runs at the microprocessor's speed, it is faster than the separate one, meaning that cache at first level is fastest and smallest. However, a separate cache is still faster than RAM.

In general, L1 instruction and data caches are up to 64 KB per core, L2 up to 512 KB per core (or shared up to two cores) and deliver data with a latency of 5 to 10 CPU cycles, and L3 may vary from 8MB to 32MB (shared across all cores or sliced to multiple instances to be associated per core) with a latency of 10 to 20 CPU cycles [7]. In some architectures, there is a Level 4 (L4) cache, which is also called DRAM buffer, offering an additional cache size of 128MB.

Hit and miss ratio are two critical definitions in cache memory since they are directly connected with the cache's performance and are defined as follows in 2.1 and 2.2:

$$Hit\ ratio = \frac{\#\ of\ cache\ hits}{(\#\ of\ cache\ hits + \#\ of\ cache\ miss)} \tag{2.1}$$

$$Miss\ ratio = 1 - Hit\ ratio \tag{2.2}$$

2.1.3.1 Cache Mapping and Optimization Techniques

Data from the main memory are brought into the cache memory using a technique called cache mapping. There are three cache mapping techniques, Direct, Fully associative, and K-way set-associative Mapping [8]. To attain an optimized usage of the cache, several replacement algorithms [9] can be implemented based on frequency, recency, frequency and recency, application, level of cache, e.t.c [10]. Cache optimization is accomplished with the reduction of hit time and miss penalty and increased cache bandwidth. A multi-level cache is used to minimize the gap between memory latency and CPU bandwidth.

2.2 Memory Hierarchy

In most systems, memory hierarchy [11] starts with a small, fast, expensive cache component, and main memory follows since it is larger, slower, and cheaper. A set of parameters are important in order to characterize the hierarchy:

- **Latency:** Time elapsed between the request of information and access to the first bit of the received data.
- **Bandwidth:** The rate at which data can be read from or stored into memory by a processor.
- **Capacity:** The amount of information in bytes the memory can store.
- **Cycle time:** The time between a read operation started and the beginning of the next one.

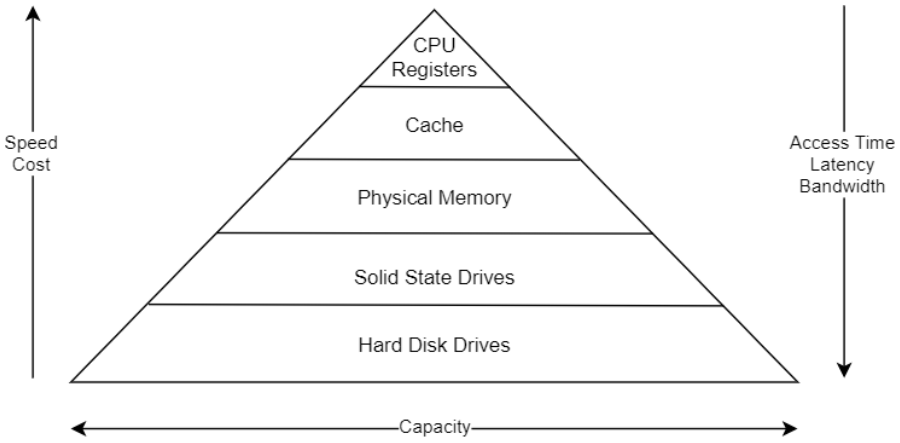


FIGURE 2.3: Typical Memory Hierarchy

Computers were initially designed without any memory hierarchy, increasing the speed gap between CPU registers and main memory due to large differences

in access time, resulting in lower performance. System performance is greatly improved by storing information into the fastest memory parts and accessing it many times before its replacement. This principle, called locality of reference, exists in two forms; spatial and temporal. Spatial locality occurs when a given address referenced before is likely to be re-accessed in a short period. In contrast, temporal locality occurs when a particular memory segment that has been accessed is most likely to be referenced next. The different levels of memory, as mentioned above, exploit both spatial and temporal locality to reduce memory access time.

In table 2.1 we present the memory hierarchy parameters found in standard systems nowadays and their typical values for various memory components.

	Access Type	Capacity	Latency (Avg)	Bandwidth
CPU Registers	Random	8∼1024 bytes	0.3 ns	System Clock Rate
Cache	Random	256 KB∼8 MB	1∼13 ns	120∼175 GB/s
Physical Memory	Random	2∼64 GB	70∼100 ns	19∼35 GB/s
Solid State Drives	Random	128 GB∼2 TB	7∼150 μs	200∼550 MB/s
Hard Disk Drives	Random	500GB∼10TB	1∼10 ms	50∼120 MB/s

Table 2.1: Memory hierarchy parameters.

2.3 Memory Management and Addressing

The CPU is not directly connected with the system's main memory but through another chip called the Memory Controller Chip (MCC). The MCC contains all the necessary logic for the read/write operation and is responsible for refreshing the memory module. During a refresh, the memory controller sends a pulse of electronic current through the RAM chips. Memory modules, such as those inside an FPGA, are volatile, meaning that they will lose all their data without constant electricity. The logical address containing the requested data issued by the CPU is converted to a Physical Address by the controller. The MCC, which is connected to the memory module by a multiplexer, locates the memory information and sends it back to the CPU.

Memory architecture consists of multiple arrays of single-bit storage arranged in a two-dimensional structure. This structure is formed by individual rows (Word line) and columns (the width of the column is called Bit Line) intersecting with each other and is often referred to as a memory bank. The banks that exist in the same chip form a bank group. Multiple chips that share the same address lines form a memory rank, and a memory module may consist of more than one

rank. For example, if a module has chips on both sides, then it is a Double Rank memory module.

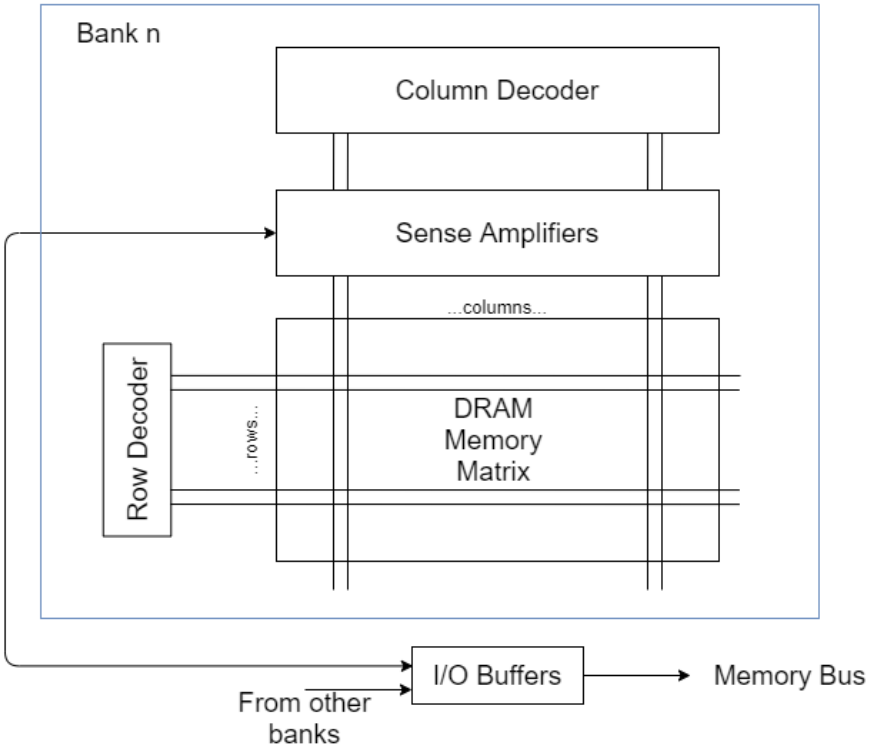


FIGURE 2.4: Detailed description of a memory's bank architecture.

Read and write operations are a 2-step process. The first step, also called RAS, selects the bank group, bank, and row. The second step, also called CAS, uses the address bits registered with the Read/Write command to select the starting column location for the burst operation. The number of bits per row indicates a memory page. In order to achieve Read/Write operations, the memory issues specific commands:

- **Activation Command (ACT):** With this command a bank opens a row for read/write operation and the row remains open until the self-refresh command. If a bank needs to access a different row the already open row must be closed.
- **Refresh Command:** In order to issue a refresh command all rows must be deactivated. After the command, all banks are in idle state.
- **Self-Refresh Command:** This command is required so that data are maintained inside the SDRAM module. This is done automatically and affects all rows in all memory banks.

- **Precharge Command (PRE):** With this command an already open row in a bank closes. This row can not be accessed again for a certain time period.

2.3.1 Memory Timings

One of the most fundamental features of memory is timing. When a request is issued by the CPU, memory requires some time in order to retrieve the requested data.

- **CAS Latency - t_{CL}:** This is the most crucial timing. It represents the time frame between sending a column address to the memory controller and the appearance of that memory's first data bit, provided that the correct row is already chosen.
- **Row Address to Column Address Delay - t_{RCD}:** The minimum time required in order to choose a row and access the columns within. The number of cycles needed to retrieve the first bit of information without having selected a row is $t_{RCD} + t_{CL}$.
- **Row Precharge Time - t_{RP}:** This timing represents the number of clock cycles between the precharge command and the correct row's opening. To read the first bit of data, provided that the wrong row is chosen, $t_{RP} + t_{RCD} + t_{CL}$ number of cycles will pass.
- **Row Active Time - t_{RAS}:** This timing represents the number of clock cycles spent between the activation of a row and the precharge command, overlapping with t_{RCD}. Since the row is refreshed, t_{RAS} equals $t_{RCD} + t_{CL}$.

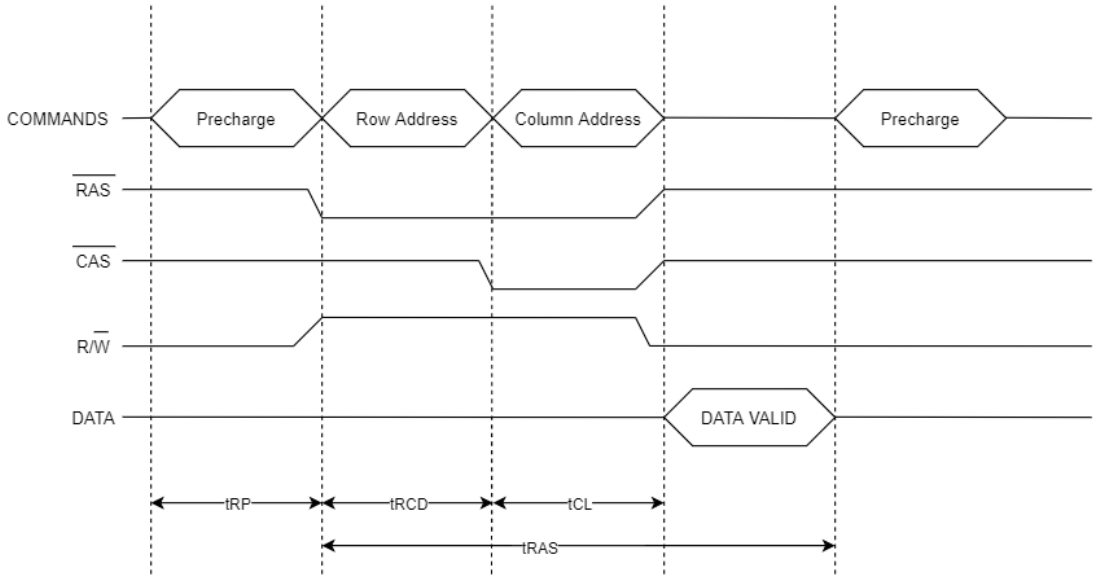


FIGURE 2.5: Simplified timing diagram of a read operation.

2.4 On-Chip Memories

2.4.1 Block RAM

Inside the fabric of an FPGA various memory elements are embedded and can be used as random-access memory, read-only memory, and shift registers. These elements are block RAMs (BRAM), Look Up Tables (LUT), Flip-Flops, Digital Signal Processors (DSP), and shift registers. BRAMs are a discrete part of an FPGA and are used to store data on the PL side of an FPGA allowing data to be transferred to the hardware accelerators (which also reside on the PL side) at much higher speeds than a DDR. On the other hand, BRAM recourses are limited; sometimes, only several KB of space is available to be used. The most recent and more expensive FPGAs come with 4-6 MB of Block RAM, since BRAM size is directly connected with the cost; larger and more expensive FPGAs include more space, providing more flexibility.

Block RAMs come in a finite size; 4/8/16/32 Kb (Kilobits) are common. They have a customizable width and depth and each FPGA has a certain amount of such BRAM modules that can be configured. The most common ways to use a BRAM are to transfer data between multiple clock domains using local FIFOs, between an FPGA target and a host processor using DMA FIFO, and between FPGA targets using peer-to-peer FIFO.

This memory module is used as Single Port, Dual Port or FIFO BRAM. The Single Port Block RAM configuration is useful when there is just one interface that needs to retrieve data. Dual Port BRAMs are available on modern FPGAs and can be used as a Memory Mapped I/O [12], providing a more comfortable

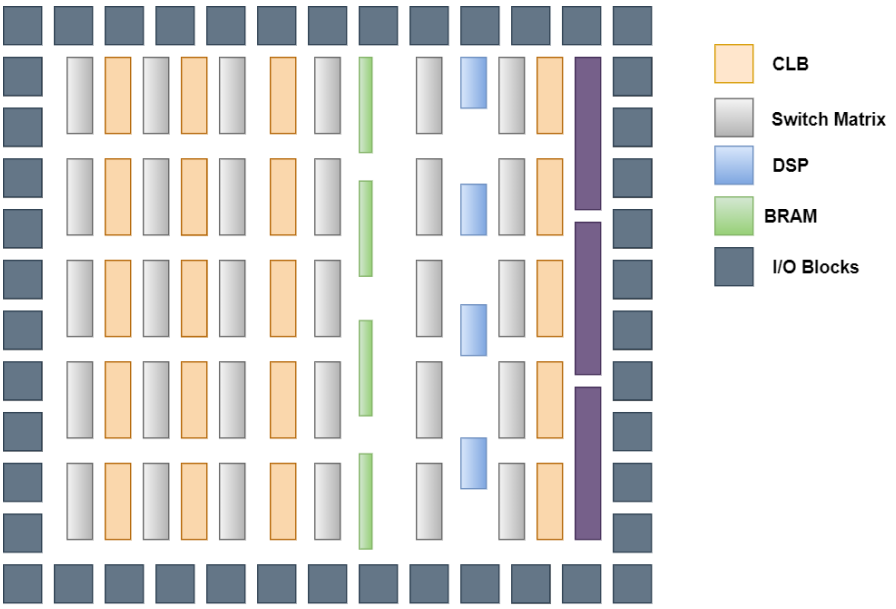
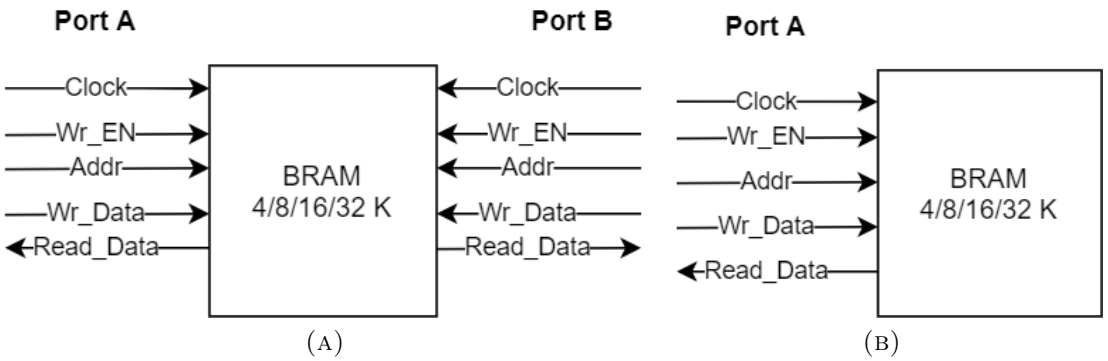


FIGURE 2.6: Part of the Logic Fabric and its constituents elements.

and low-cost solution. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations. Finally, in the FIFO configuration, one port is used to write data and one to read them.



(A) (B)

FIGURE 2.7: Dual Port (A) and Single Port (B) BRAM.

2.5 Theoretical background sources

A considerable part of the information presented above was acquired from the Computer Architecture and Reconfigurable Computing Systems courses of Electrical and Computer Engineering school at the Technical University of Crete. Furthermore, the book *Fundamentals of Computer Organization and Architecture* [11] was an insightful guide and provided useful knowledge concerning memory organization and computer systems. Moreover, the book *Exploring Zynq MPSoC With PYNQ and Machine Learning Applications* [13] provides an extensive analysis of

the ZYNQ FPGA family which is evaluated in this thesis. Finally, the doctoral thesis of Dr. Aggelos Ioannou [14] provided useful spot-on information about memory components and their purpose and acted as a guide, especially in the early stages of this thesis.

Chapter 3

Related Work

In this chapter, we present some of the most notable scientific contributions related to our research. Most of the work presented here is based on published papers.

3.1 Memory Allocation techniques on FPGA platforms

Nowadays, FPGA's have limited on-chip memory resources, which makes efficient handling of those resources vital. FPGAs have been used in high-level image processing due to their power efficiency and customization abilities. Relatively simple algorithms that perform local region operators (sliding window filters [15], for example) can be implemented as line buffers. On the other hand, algorithms that use global operators require complete frames to be stored in situ (identification, object detection, etc.).

Processing-specific architectures have shown inadequate BRAM utilization resulting in various architectures such as the mirroring memory system [16], which prohibits scaling for higher frame sizes, the neighborhood loader [17]which does not support random access, and the spatial parallelism [18] which does not support real-time streaming. Off-chip memory access by storing image frames into peripheral memory components solves the capacity barrier but decreases performance. Parallel matching arrays [19] for accelerating computation can hold only one row each time; the remaining frame is still stored off-chip. Vector memories for accelerating vector processing [20] on FPGA rely on random access to peripheral memory components.

3.1.1 Memory Partitioning and Mapping scheme

The work of P.Garcia, D. Bhowmik et al. [21] provides insight into partitioning image frames into BRAMs to minimize the number of required on-chip memories. Assuming a BRAM of C storage capacity and i the number of possible BRAM configurations then we have a vector Cfg of i elements such as :

$$Cfg = \begin{pmatrix} (M_1, N_1) \\ (M_2, N_2) \\ . \\ . \\ (M_i, N_i) \end{pmatrix}$$

where M_i, N_i are the width and height of each BRAM configuration and only one pair of M_i, N_i is non-zero. Their work focuses on image frames; 3-D arrays are presented as 2-D arrays of Width (columns), Height (rows) dimensions, and each cell of the 2-D array represents the pixel bit width, B_w. Their goal is to find the appropriate amount of BRAMs needed, which are represented as a 2-D array of BRAMs. Given a 2-D image frame of dimensions $x = B_w$, $y = W_{image} \times H_{image}$, a partioning scheme which assigns pixels accross $a \times b$ BRAMs is given by:

$$p(x, y) = Cfg * ((a_1, b_1), ..., (a_i, b_i)) \tag{3.1}$$

where * stands for linear combination and only one (a_i, b_i) pair has non-zero components. Finally, they map the 2-D array into row-major or column-major 1-D arrays. Their goal is to maximize utilization efficiency using these two formulas:

$$Maximize\ E = \frac{x \times y}{a_p \times b_p \times C} \tag{3.2}$$

$$((a_p \times M_p \geq x) \cap ((b_p \times N_p \geq y) \tag{3.3}$$

where $x = B_w$, $y = W_{image} \times H_{image}$, a_p and b_p are the array of BRAMs configurations, C is the BRAM capacity, $M_p = BRAM_{height}$ and $N_p = BRAM_{width}$.

For example, if $(x, y) = (8, 76800)$, then for a Xilinx Virtex 7 BRAM configuration of $(M_p, N_p) = (1, 16384)$ the BRAM usage count is 64 ($(a_p, b_p) = (8, 8)$), which is the minimum number of BRAMs needed in order to store the image frame and achieve maximum utilization efficiency. Their experiments showed that partitioning data unevenly (meaning that $a_p \neq b_p$) utilized less amount of BRAMs and increased utilization efficiency. Furthermore, they addressed the power implications of each BRAM configuration. BRAM static power is directly proportional to utilization and increased per read/write operation. Their results show that minimizing the horizontal usage of BRAMs (meaning that BRAM width is increased but the number of BRAMs in the x axis is decreased) is more suitable for clock gating. Their algorithm selects the optimized utilization solution by iterating over wider BRAMs. They implemented frame buffers in Verilog, creating BRAMs according to the desired configurations and compared them with strategies employed by commercial HLS tools. Their experiments included a sequential read of

the whole frame and a sliding window of 3×3 read of the frame for monochrome and RGB images.

The results revealed a reduction in BRAM utilization of up to 41.4% for both monochrome and RGB images when using their hand-coded modifications and appropriate BRAM configurations (BRAM width, height, and array BRAM width, height) comparing to Vivado HLS results. Static power consumption was slightly higher than the Vivado HLS. Albeit, their method showed excellent results in total dynamic power consumption (in sequential and sliding window experiments) and BRAM power consumption.

3.1.2 DOMMU architecture

Dynamic memory management is crucial and enhances reconfigurable computing gains since it can avoid static memory allocations when implementing a design. Ghada Dessouky, Michael J. Klaiber et al. [22] proposed a Dynamic On-chip Memory Management Unit (DOMMU) which targets the dynamic management of BRAMs during run-time. Dedicated hardware solutions have presented excellent performance but low flexibility and run-time adaptivity. Their work focuses on four critical topics as presented below:

- **Dynamic Memory (De) Allocation**: Since static memory allocation reserves enough BRAM to cover worst-case scenarios, previously allocated memory regions remain unused. Unused memory segments can be shared and allocated through different PEs. This dynamic sharing and allocation reduces memory requirements and improves BRAM utilization; allocation must occur faster than access to the PE to ensure that data access is correct and updated.

- **Transparecy**: DOMMU's configurations have to be identical to BRAM accessing, achieved by virtual address mapping whilst maintaining single-cycle latency.

- **Scalability**: DOMMU's design accepts user configurations in order to be scalable, concerning the number of BRAMs, BRAM's number of ports, and types. Furthermore, the unit provides support for BRAM sharing between PEs by using dual-port BRAM access.

- **Optimal point in design**: Bandwidth, latency, and hardware resources are concerns that the unit must meet. Dedicated channels between PEs and their corresponding BRAMs assure latency of one clock cycle; hardware resources must be kept to a minimum by dynamically reserving and deallocating memory regions.

Each PE is assigned with one or more memory ports that interface with DOMMU for BRAM allocation/deallocation and access in their architecture. Each PE holds information regarding BRAM configurations, which BRAM is assigned to the PE, the frequency of accesses, and the amount of BRAM required by each PE. BRAM elements are arranged as logical pages that are assigned to memory ports. Logical pages are assigned up to N BRAM elements, which in turn acquire a logic identifier. At runtime, the logical identifier is associated with a physical identifier so that each memory port knows its logical page. Since the PEs communicate with BRAMs by logical addresses, each memory port is assigned a translator which performs the aforementioned procedure.

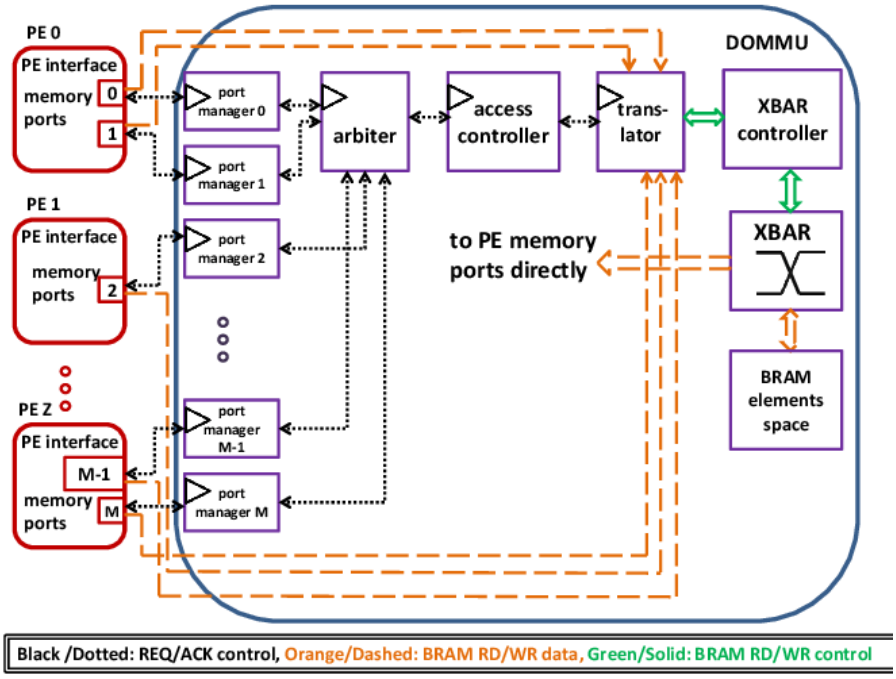


FIGURE 3.1: DOMMU Architecture. Copyrights to:[22].

The XBAR switch allows bi-directional communication between BRAM and PE to support read/write operations. The access controller receives and handles BRAM (de)allocation requests and is implemented as an FSM.Every memory port has a priority assigned either at design-time (which remains unchanged) or run-time(which can change according to needs). The arbiter selects which memory port to serve in every cycle, according to their priority. Dynamic priority is essential in real-time systems since some applications are more time-critical than others. Every PE memory port consists of a dedicated BRAM port, which supports read/write operations and a control port. The control port is assigned a memory manager that matches the requested BRAM type, width, height to the closest BRAM configuration.

This architecture allows for additional BRAM space to be dynamically allocated when the memory port's BRAM is wholly utilized. Their results showed that

for a configuration of 8 memory ports and 40 BRAMs, LUT utilization reached 17% (9% was associated with the XBAR switch) of available resources in Xilinx Virtex-5 LX110T. Increasing the number of memory ports or BRAMs, they observed a linear increase in resource utilization since the XBAR switch crosspoints were also increased as well as the number of arrays required to handle more memory ports or BRAM elements. To be more specific, increasing the number of BRAMs has less impact on utilization than increasing the number of PEs. Increasing the number of memory ports or BRAM elements slows down the design, which is operated at a maximum of 140 MHz. It is worth mentioning that a typical frequency of Virtex-5 FPGAs is 450 MHz.

Finally, to test their dynamic allocation design and compare it to static allocation, they used a CCL [23] algorithm that labels image pixels into independent components based on pixel connectivity. Their architecture proved that only 3% of the static allocated memory is needed, providing them with a factor of 33 in BRAM utilization.

3.1.3 Multi-ported Memory on FPGA

Multi-ported memory architectures are often used in FPGA designs, offering high bandwidth and demanding more BRAM space, which results in BRAM lacking for other parts of the design. Several techniques implement multi-ported memories on FPGA. Replication of data into multiple memory banks so that the architecture can support multiple concurrent reads and writes (when a write occurs, data in all memory banks are updated) is a simple, logic-free solution requiring more BRAM space directly proportional to the number of memory banks. Using Live Value Table as a technique, the architecture supports multiple writes into different memory banks simultaneously. However, extra logic is needed in order to select the appropriate memory module for the read operation. Another approach is the XOR-based architectures, which implement the XOR of two instances of the same location to return the most recent data as described in 3.4. This method achieves higher bandwidth (since it does not use the logic the LVT method does) but increases memory utilization since it adds more memory ports. A more detailed presentation of different multiported memory designs is given in [24].

$$R_0 = (A_{new} \oplus A_{dirty}) \oplus A_{dirty} = A_{new} \tag{3.4}$$

In [25] an XOR-based architecture is proposed in order to implement multi-ported memory on FPGA. They implement a 2R1W (2 reads, one write) memory structure that can support multiple reads/writes using multiple parallel-access banks. They use four memory banks and one XOR bank, which stores the XOR-ed value of the data with the same offset in every memory bank. If two reads

target the same bank, then the first one will gain access to the memory bank while the second one will XOR the data at the same offset of other banks and the XOR-bank. The write operation includes two steps; firstly, the data are written directly to the appropriate bank, and data at the same offset in different banks are read and XOR-ed. Secondly, the XOR-ed value is stored back at the XOR bank ($total\_memory\_depth/number\_of\_banks$ in size).

By implementing more 2R1W blocks, they acquired more read ports without slowing the design. To support more write operations, they use multiple banks, a remap table, and an additional bank that keeps the latest data written. If two writes occur at the same bank, then the first one will be written directly to the corresponding memory address. In contrast, the second one will be written at the additional bank at the same offset, and the memory map will keep the location of the second write. This architecture requires less memory space than the LVT but utilizes more registers.

In their experiments, they used a Virtex-7 FPGA and implemented a 4R2W memory architecture. Their results showed a 37.5% reduction in BRAM usage compared with replication techniques and a 25% BRAM usage reduction compared with LVT. However, they experienced a slight frequency decrease in their design compared to the aforementioned methods.

As it is clear from the previous chapter, BRAMs are not multi-ported by default; they have only two ports and can perform 2 read and 1 write operation per cycle. Today's designs contain multi-ported BRAM modules that are created by combining many single or dual ported BRAMs. 2W2R (2 writes/ 2 reads) BRAM combinations provide multiple read/write ports but data are fragmented; every BRAM module contains its own information and data can not be shared throughout the BRAM modules. In [26] an n bi-directional port multi-ported memory design using BRAM modules is presented. They offer an almost logic-free design, implementing multi-ported BRAM modules. They use dual-port BRAMs that are organized in memory banks controlled by decision making modules. These modules store information about the last written data and provide their contents when a read operation is executed. Figure 3.2 presents the architecture of the n bi-directional multi-ported memory design.

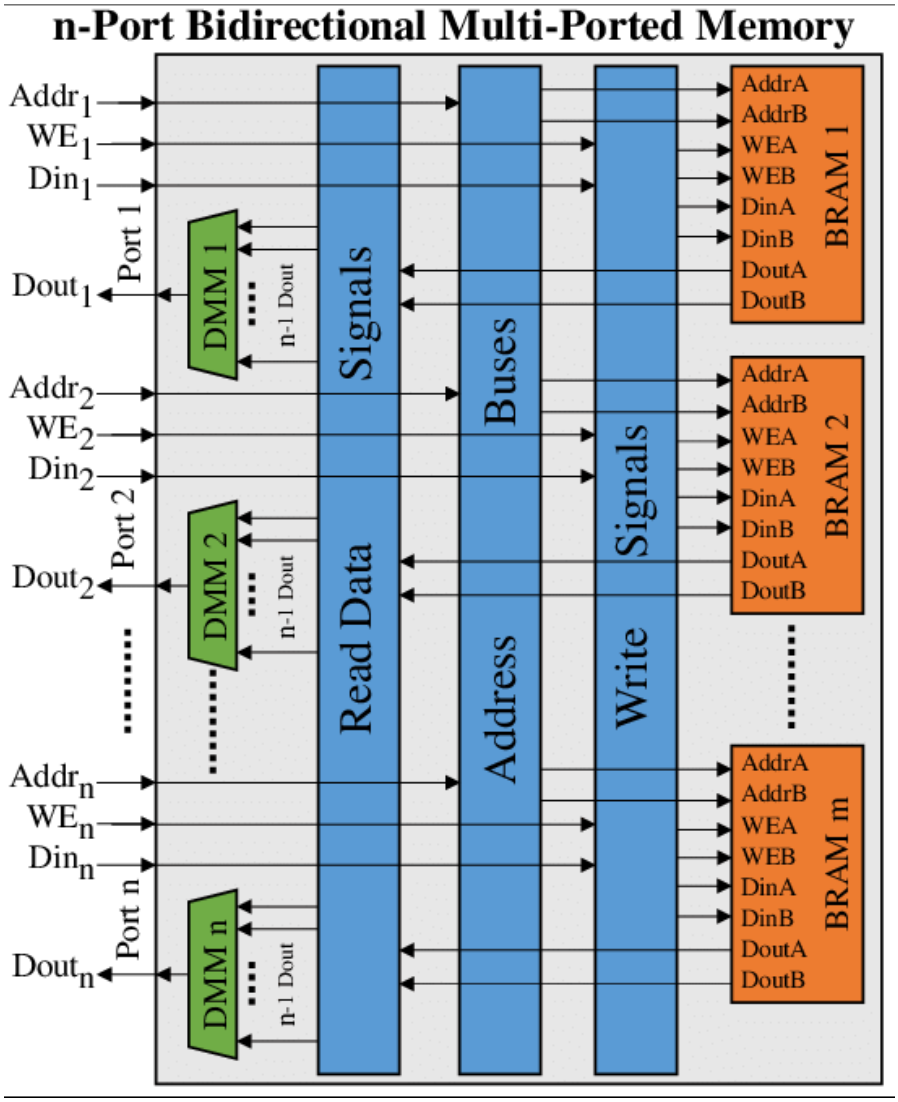


FIGURE 3.2: Architecture of the n bi-directional multi-ported memory design. Copyrights to:[26].

3.1.4 Memory Partitioning for Multidimensional Arrays

Data partitioning is an expensive task, and FPGAs are used in order to accelerate this procedure [27]. FPGAs provide enough computational power and are extensively used in parallelized designs. High-speed data streams are required in order to supply computational units, and many applications require multiple access to arrays in a single cycle using one memory port, requiring pipelining execution, and memory partitioning architectures [28], [29]. Methods for partitioning data structures into multiple memory banks for increased parallelism and performance are presented in [30]. As we mentioned before, memory partitioning algorithms target 1-D arrays. Multidimensional arrays (image, video) are flattened before a partition takes place. In [31] the MemGuard architecture is presented, a memory bandwidth reservation system that supports efficient memory performance isolation on multi-core platforms. The work of Yuxin Wang et al. [32] provides a

partitioning algorithm for linear transformation of multidimensional arrays, an offset generator scheme, and a heuristic solution based on memory padding. Data elements are located on new memory banks after memory partition takes place based on the following formula:

If M is the data domain then $\forall d_1, d_2 \epsilon M$

$$d_1 \neq d_2 \Leftrightarrow (f(d_1), g(d_1)) \neq (f(d_2), g(d_2)) \tag{3.5}$$

where $f(d), g(d)$ are mapping functions for the bank number and the bank offset, respectively. They compared their architecture with state-of-the-art 1-D array flattening techniques using Richian-denoise algorithms, a Sobel edge detection algorithm, and different loop kernels of motion compensation from an H.264 decoder. Their results showed a 21% reduction in BRAM utilization, 19% reduction in slices, and 46% reduction in DSPs. However, they experienced a small increase of 0.6% in CP. The bank count decreased with their proposed method, but power consumption was higher (when compared to the power consumption of flattening techniques) in almost every experiment.

3.1.5 Memory evaluation and architectures in stencil computing

Stencil computing is used in a wide range of applications, from physical simulations to machine learning. Their computing time is occupied by the kernels of stencil computation, and acceleration techniques are needed. In [33], a scalable streaming array (SSA) is designed using multiple FPGAs. They also create an in-depth pipelining approach over consecutive iterations achieving linear scalability for multiple devices and constant memory bandwidth. In their experiments, they used 2-D and 3-D Jacobi computations, a Master FPGA of 128 PEs in both cases, and a slave FPGA with 128 and 120 PEs for the 2-D and 3-D case, respectively. They achieved a peak performance of 297.5 GFlops/s for the 2-D and 280.9 GFlops/s for the 3-D Jacobian computation while maintaining a 2 GB/s memory bandwidth. Finally, they measured that the power consumption of the SSA for the 2-D and 3-D case is 1.3 GFlops/sW and 1.07 GFlops/sW, respectively. They also experimented with other FPGAs and estimated that 100 Stratix V FPGAs achieve sustained performances of 17 TFlops/s for 2-D and 17.7 TFlops/s for 3-D Jacobian computations. The work mentioned in [34] represents an extensive analysis of stencil optimizations and performance modeling on microprocessors, experimenting with various algorithmic approaches and architectural platforms.

The work of Konstantinos Kalaitzis et al. [35] focuses on the memory performance of a hybrid supercomputer such as the Convey HC-x in memory patterns

found in stencil computations where a large amount of data is needed from external memory. This HRPC technology includes a scalar instruction set, coherent cache connection to the host processor, and a high bandwidth local memory system.

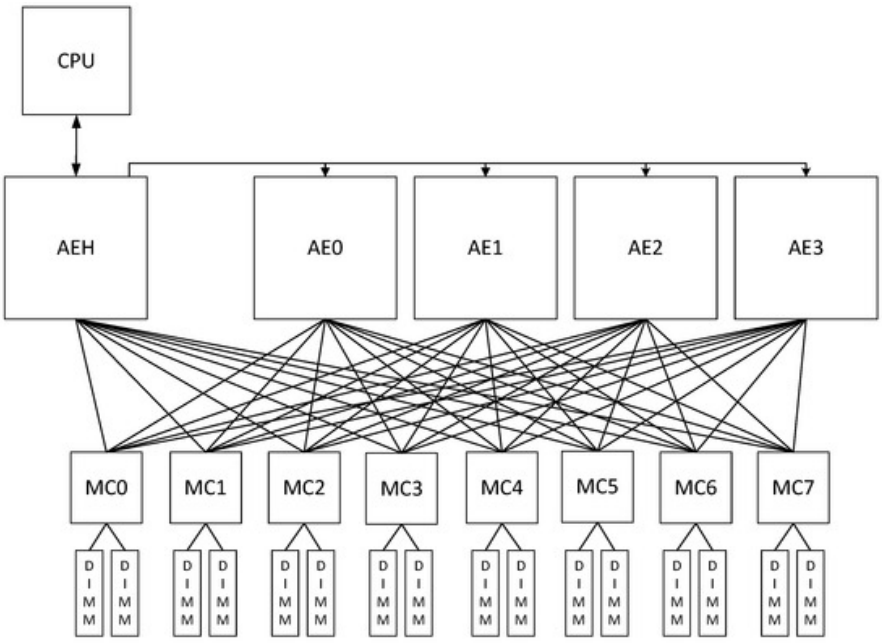


FIGURE 3.3: Convey HC-x Architecture. AEH: Application Engine Hub, MC: Memory Controllers, AE: Application Engine. Copyrights to:[35].

Their design frequency was set at 150 MHz, and their focus was centered around bandwidth and latency measurements of the Convey HC-x memory subsystem. Latency is measured by when the AE sends a read request and the time data reach the AE. Their results showed 120 clock cycles on average. As it was expected, when the read address is between 8 to 64 bytes, the variance of latency is low, whereas when data are located in other DIMMs and served by a different port, latency is increased. Furthermore, they concluded that increasing the number of memory controllers raises the variance of latency. Since Convey architecture is designed for a considerable amount of requests from memory, this latency is not a barrier; MCs hold the whole block of each DIMM used each time, meaning that consecutive requests that concern data in the same memory space are served quickly. Bandwidth was measured at 1.2 GB/s when only one memory port is used, and data are in burst mode and increases as MC ports increase, reaching 9.6 GB/s for 16 MC ports. Their results did not match the theoretical expectations since each memory controller port did not have a dedicated FSM that sends requests independently. They also experimented with data strides concluding that in strides that get mapped into different DIMMs, the performance is identical to sequential memory bursts. In contrast, in strides that are mapped into the same

DIMMs, the performance decreases. Finally, they experimented with irregular data accesses using Zuker's algorithm reaching a bandwidth of 1.2 GB/s.

3.2 Thesis Approach

This thesis goal is to evaluate the main memory module's behavior using existing IP cores provided by Xilinx that simulate various Processing Elements (PEs). We evaluate an existing FPGA's memory behavior, providing the foundations of a platform that can be easily scaled up to be used in more advanced FPGA systems (for example, FPGAs with more ports at the PS side) with minor modifications. We also aim to categorize memory access methods by determining the optimal transfer data size for each subcategory. This work can be used in the future alongside hardware accelerators in order to expedite their process further.

For the purposes of this thesis, we tested how memory behaves in sequential and random access. To be more specific, 1-D CNNs, such as the one presented in [36] require data that reside in consecutive memory addresses and can be efficiently accessed with the use of DMAs. Data are stored in sequential addresses in memory and can be obtained with bursts without accessing different rows every few cycles. Algorithm 2 in Chapter 5 simulates these applications where multiple processing elements (PEs) require data that can be obtained without strides in memory. For small burst sizes (meaning that the network requires a small amount of data in order to produce a single output), both PEs access the same row. On the other hand, multi-dimensional CNNs require data that reside in different rows accessed by striding in memory. Algorithm 3 in Chapter 5 simulates a multi-dimensional CNN memory access pattern by creating requests that correspond to different layers of a neural network.

CNN	**Minimum #params**	**Layer**	**Maximum #params**	**Layer**
AlexNet	34.944	Conv1	37.752.832	FC6
VGGNet16	1.792	Conv3-64	102.764.544	FC
ResNet18	9.472	Conv1	2.359.808	Cov5-(2-4)
GoogLeNet (Inception v1)	4.160	Conv1x1	1.353.968	Inception (5b)

TABLE 3.1: Various multi - dimensional CNNs and their maximum/minimum parameters.

Table 3.1 depicts the maximum/minimum number of parameters (and the corresponding layer) for four CNNs : AlexNet [37], VGGNet [38], ResNet [39], and GoogleNet [40]. Given a representation of single-precision floating-point, VGGNet

requires 7.1 KB of data to complete Conv3-64 layer and 411 MB to complete FC layers. Since we have limited on-chip memory resources we can safely assume that each request will not be bigger than 1-2 MB (2 MB is the worst-case scenario, most of the time requests are in hundreds of KB and less even for larger layers). Having this information as a starting point, we chose different burst and memory stride sizes to simulate as accurately as possible the memory requirements of various CNNs.

Chapter 4

Tools Used for FPGA Implementaion

The architectures presented in Chapter 5 were implemeted using the Xilinx Vivado Design Suite - HL System Edition 2019.2 [1] developed by Xilinx for synthesis and analysis of HDL designs, written in VHDL or Verilog. It is superseding Xilinx ISE [2] and it can sypport System-On-Chip designs providing additional features. The tools we used are the Xilinx Vivado IDE, Xilinx Vivado HLS, and Xilinx SDK.

4.1 Vivado IDE

Xilinx Vivado Integrated Design Environment (IDE) is the GUI for the Vivado Design Suite. Its main feature is the Intellectual Property (IP) Integrator, which allows its users to instantiate and connect different IP cores, either using its GUI interface or using Tool Command Language (Tcl), a programming interface. The Tcl commands can be written in the Tcl console in the Vivado IDE or using the Vivado Design Suite Tcl Shell. IP cores can be designed using Vivado HLS, or they can be a part of the Vivado IP catalog. The IPs can be graphically connected and configured using Vivado's GUI interface, and the process is accelerated with Vivado's auto-connect feature. Vivado IDE can synthesize, implement, place and route designs written in HDL languages like Verilog or VHDL containing IP blocks created using Xilinx HLS for C/C++ designs.

The design process begins by adding all the necessary IPs to the design and connecting the appropriate ports by hand or using the auto-connect feature. Every module in the design has a clock and reset signal, and many modules have a base address assigned to them. After the design's validation is complete, the project is synthesized; the RTL design is turned into a logic gate schematic. A successful synthesis means that implementation can occur; the placing and routing of the synthesized design in an FPGA of our choice. After completing these two

[1] https://www.xilinx.com/products/design-tools/vivado.html
[2] https://www.xilinx.com/products/design-tools/ise-design-suite.html

steps, the user is provided with a report[41] containing critical warnings, messages, and/or errors. The most important information presented in this report concerns timing, utilization, and power. Timing information is critical since it estimates the net delays based on connectivity and fanout. The utilization report presents the design's utilization based on the resource type (Flip- Flops, BRAMs, LUTs, and DSPs). Finally, the power report depicts the estimated power consumption and the thermal effects of the device's design. If the implementation results match our expectations, then the bitstream file is generated to program the FPGA. An essential feature of the Vivado IDE, which was extensively used in our experiments, is the Integrated Logic Analyzer, which can be used to record signal values between IP cores and deploy triggers that notify the user when something specific happens.

4.2 Vivado HLS

Xilinx's Vivado HLS is used to synthesize programs written in C, C++, OpenCL, and SystemC. This tool allows the user to target specific FPGAs and create a design without an RTL schematic. The user can write functions in their chosen language, and HLS synthesizes them into IP blocks by generating low-level VHDL or Verilog programs, according to features and constraints set by the user. The primary purpose of Vivado HLS is to optimize code generated for hardware platforms and provide the user with the capability to create their custom IP. The generated blocks can be integrated into hardware systems using the Vivado IDE.

The user creates a project with all the necessary libraries and designs the IP block according to their needs. The tool accepts, most of the time, non-optimized code and adds constraints to the exported hardware-specific IP block. The directive's purpose is to implement an optimized code, direct the synthesis process to implement a specific behavior, and maintain a high resource management level without changing its function in the project's simulation part. Moreover, clock uncertainty, FPGA target, and clock period are added as constraints to the synthesized IP block. To further optimize the code, pragmas can be used as commands to modify the generated code. Finally, the user can write a test bench in C/C++ language to test the written code and identify errors. The exported IP block is verified by a process called C/RTL Cosimulation. The testbench is used once again, but this time the function is replaced with the exported IP.The last step is to synthesize the written code and produce the VHDL/Verilog function. The tool allows the user to directly write in VHDL/Verilog or edit the generated HDL code. In this case no synthesis step is required.

4.2.1 Vivado HLS Synthesis Report

After synthesis is complete, the user is presented with a report containing the information presented below. The two most important reports are timing and utilization.

Timing presents information about the design's clock frequency and latency estimations per loop per module for the entire design.

- **Latency:** The number of cycles required for the function or loop to complete all output values.
- **Initiation interval (II):** Number of cycles need in order for a function to accept new data.
- **Loop iteration latency:** The time required by a loop to complete its iteration.
- **Loop initiation interval:** The number of clock cycles before the next iteration of the loop starts to process data.
- **Loop latency:** Number of cycles to execute all iterations of the loop.

The utilization report depicts the percentage of resources of the selected device that are utilized in our project.

- **Area:** This metric presents the percentage of LUTs, BRAMs, FFs, and DSPs used in our design. Every hardware component (Multiplexers, FIFOs, Instances, Memories, Registers) type requires resources mentioned in the utilization report.

4.2.2 Optimizing the design in HLS

The UG902 [42] reference guide from Xilinx provides detailed and insightful information on optimizing the design in Vivado HLS. As we mentioned before, directives are optional, and they aim to reduce latency, area, and resource utilization and improve throughput. The directives can be added as pragmas into the input code or in multiple solutions. Each solution may contain a different set of directives providing a better understanding and allowing the user to experiment and find the optimal solution. Xilinx provides us with 24 optimization directives, and we present the ones used frequently.

- **Allocation**: This limits the number of operations, cores, or functions used and forces the sharing of hardware resources resulting in a latency increase.

- **Array Map**: Multiple smaller arrays are combined into a single large array in order to reduce BRAM usage.
- **Array Partition**: Large arrays are partitioned into multiple, smaller ones or individual registers. This improves data access and removes bottlenecks caused in BRAM. Arrays are, by default, implemented as a single port BRAM unit. Partitioning arrays increases the number of read/write ports allowing parallelism. Albeit, memory instances are increased, but throughput is improved in these applications.
- **Array Reshape**: The specified array is reshaped into a greater word-width array, improving block ram accesses without utilizing more BRAM.
- **Interface**: Specifies each argument's port type. The arguments of the top-level function are mapped into RTL ports.
- **Latency**: Sets a max and min latency constraint.
- **Loop Flatten**: Perfectly nested loops collapse into a single loop with improved latency.
- **Loop Merge**: Overall latency is reduced by merging consecutive loops. Logic optimization is improved, and sharing is increased.
- **Stream**: Specifies that a specific array is to be implemented as a FIFO instead of the RAM channel initially implemented. When using hls::stream, the STREAM optimization directive is used to override the configuration of the hls::stream.
- **Top**: The top-level function for synthesis is specified in the project settings. This directive may be used to specify any function as the top-level for synthesis.
- **Unroll**: Unroll for-loops to create multiple instances of the loop body and its instructions that can then be scheduled independently.

In addition to the directives mentioned above, Vivado HLS provides a number of configurations that are used to change the default behavior of the synthesis. The user can configure how the arrays are partitioned, the default memory channel and FIFO depth, the I/O ports not associated with the top level function arguments etc.

4.3 Vivado SDK and Xilinx Vitis IDE

Software Development Kit (SDK) is an IDE tool used to develop embedded software applications that target Xilinx's ARM processors, like Zynq UltraScale+ MPSoC, Zynq-7000 SoCs, and the Microblaze microprocessor. SDK is an Eclipse-based application, including a C/C++ editor and compiler, build configurations, and automatic Makefile generation. Debugging and profiling software code is also available through the SDK tool. In Vivado Design Suite version 2019.2, SDK is unified with SDSoC and SDAccel into Vitis Unified Software Platform. Applications can run on the ARM cores either on the MPSoCs or external to the FPGA. The applications created in this tool act as a director and configure the targeted FPGA. Applications like data transfers to and from storage devices or Ethernet to and from other memory components (sometimes located on the PL side), initialization of IPs, DMAs, and other instances, are examples of the tool's usage. Moreover, bare-metal applications can utilize multi-core processors that run concurrently using a scheduler. Embedded Linux applications can be built on Linux operating systems using Petalinux tools [3]. Since it interfaces directly with Vivado IDE, useful tools like the Xilinx Software Command Tool (XSCT) [4].

SDK can be launched through Vivado IDE or as a standalone application. After a successful implementation, the generated hardware is exported, including the generated bitstream, used to program the target FPGA. SDK imports the generated hardware wrapper and creates the Board Support Package, which includes all the suitable device drivers of the hardware design and several libraries that can configure the FPGA. The SDK prompts two pages every time a new project is created. The system.mss page contains information about the target device, operating system, peripheral drivers, and libraries included in the BSP. The second one, lscript.ld (Linker Script) is used to control where different executable sections are placed in memory. Thus the user can define new memory regions, change the assignment of memory sections, and alter stack and heap sizes. SDK offers the ability to launch an application on an FPGA using System Debbuger, enabling excellent debugging features. One of them, which was extensively used in our experiments, is the memory view. This feature provides the user with a large array; every cell corresponds to an address of the available memory space. The user can access both PL and PS addresses and validate that data are written in the correct memory sections.

The SDK tool can create three types of applications. First Stage Boot Loader (FSBL) is an application that resides in the root folder of the board's primary

[3] https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[4] https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/xsct/intro/xsct_introduction.html

storage device (SD card in our example). These files are read during the boot of the system, triggering a boot loader sequence. Most of the time, switches must be configured in order for the board to read the FSBL files. In this case, the target FPGA can be programmed through the primary memory storage (our case) and not using a JTAG cable. Bare-metal applications are utilized through the SDK, which programs the PL part of the FPGA and are loaded into the ARM core using the JTAG port. Finally, Linux applications require a Linux operating system to run on the processor. The Linux operating system is loaded through the primary memory storage and can program the FPGA through the console window.

The Hardware manager tool of the Vivado GUI can be used to program the PL side directly. The PL side can also be programmed through the SDK by setting the path to the bitstream file in the Run Configurations window. Vivado's IDE Hardware Manager connects to the design's ILAs, providing the user with real-time monitoring. Finally, the SDK input/output values can be given/read through a console window using the UART port in all application types.

Chapter 5

Memory Subsystem Evaluation

A balance between performance and cost can be provided by Multi-Processor System-on-Chip (MPSoC) since they feature a plethora of memory components, such as DRAMs, Block RAMs, and multiple levels of cache. Studies that target the various subsystems of those FPGAs are vital in order to increase their computing abilities and contribute to more efficient use.

Optimizing data transfer speed is crucial since, in many applications, the execution time of an algorithm implemented in a hardware accelerator is less than the time required for the I/O to supply the data to the PL part of an FPGA. A typical neural network consists of millions of parameters needed in order to compute the result. A typical FPGA has enough GB of D-RAM to store these data but has limited bandwidth and just a few MB of B-RAM, which is extremely fast. In our case, we have 62 million (64-bit double-precision floating-point) numbers that represent the weights of a neural network. This means that we have 248 MB of input data (without considering the MB of image data) that can be easily stored in DDR on the PS side of the FPGA.

In this chapter we present in detail the FPGA platform used in our experiments describing all the necessary systems and peripherals that are used and evaluated. We describe how data are obtained from the SD card and why they need to be stored in the external main memory module. Furthermore, we evaluate the DDR4 module using experiments that simulate different access patterns of various CNNs. At first, we evaluated the DDR alone to test its behavior in sequential and random access patterns from single access (retrieving few bytes of data) to row-bursting (retrieving few KB or MB of data). Since different applications require different access patterns we simulate applications that require both small and large amount of data, that could be multidimensional. We measured the system's performance using only master AXI (described in section 5.1.2) ports to transfer data from the DDR to the BRAMs on the PL side using ready-made IP cores provided by Xilinx.

5.1 FPGA Platform

The architectures implemented and the evaluation process targets Xilinx ZCU102 [1] evaluation board.

Processing System		**Programmable Logic**	
APU Freq.	1.5 GHz	**Flip -Flops**	548160
L1 cache	32 KiB	**Block RAM**	4 MB
L2 cache	1 MiB	**Logic Cells**	599550
PS DDR	4 GB	**LUTs**	274080
DDR throughput	$2400MT/s \times 64 - bit$	**DSP Slices**	2520
		BRAMS	912
		PL DDR	512 MB

TABLE 5.1: ZCU102 block features and memory sizes.

This specific FPGA is a general-purpose evaluation board for rapid prototyping based on the Zynq UltraScale+ XCZU9EG-2FFVB1156E-2-i MPSoC with Cortex -A53 + R5 processing system. High-speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, various peripheral interfaces, and FPGA logic for user-customized designs provide a flexible prototyping platform. The board uses a Quad-core Cortex-A53 MPCore as an APU, a Dual-core Arm Cortex-R5 as an RPU, and an Arm Mali-400 MP2 as a GPU.

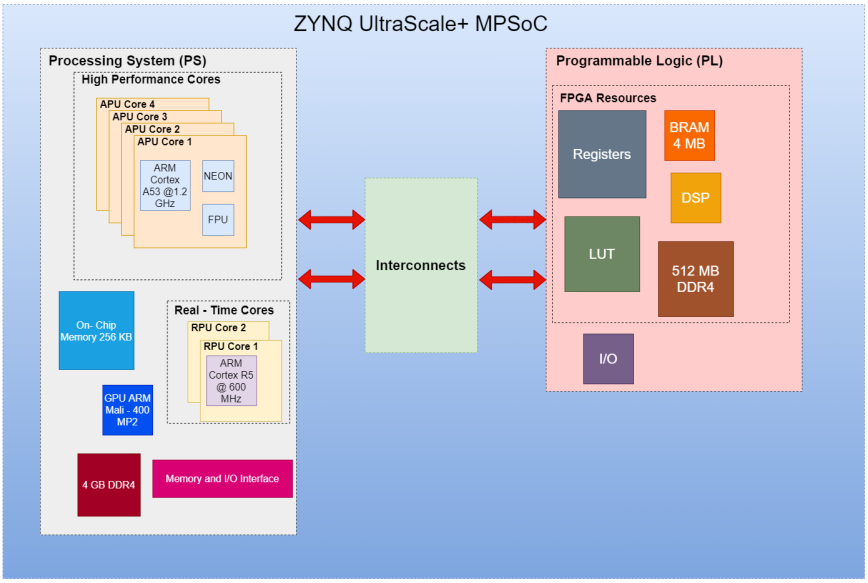


FIGURE 5.1: Simplified Block Diagram of the UltraScale+ MPSoC Architecure.

[1] https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

5.1.1 I/O

In order to test and implement different data transfer methods, we have to understand how the PS↔PL data transfer is achieved. After passing the data from the DDR of the processing system, we have to choose which of the three following categories is the most efficient one, based on our application.

Memory Mapped I/O

This method performs the I/O connection between the PS and PL using the same global address space for the memory (DDR) and the I/O extensions, mapping every component to its own address. It is observed that even though the implementation part is relatively easy, we have to pay almost 50 clock cycles per request to initialize the DDR when accessing it randomly. Sequential access, on the other hand, seems to have an advantage when using this method. Furthermore, write/read operations always need to include an address in order to be completed (the master must send the address to read/write the data to the slave).

AXI-4 Stream Interface

This method provides the opportunity to create a continuous communication, in a FIFO form, between the DDR and the PL side components. This technique hides the DDR initialization cost, and components are connected through a channel using the DMA IP (described in section 5.1.4) by Xilinx.

BRAM

In this method we use BRAM IPs (described in sections 5.1.8 and 5.1.9) to transfer data from the DDR to on-chip BRAM memory modules. Data can be transferred with or without a DMA(described in section 5.1.4) (it depends on the amount of data we need to transfer). BRAM offers huge bandwidth opportunities, but the size of the components is relatively small (only a few MB). This method gives the user the opportunity to choose address, registers and BRAM areas. However, it is observed that the BRAM controller itself has to be instantiated in the design, which means that BRAM utilization increases.

5.1.2 AMBA - AXI4 Interface Protocol

AXI [43] (Advanced eXtensible Interface) is part of ARM AMBA, a family of microcontroller buses, and almost all Xilinx's IPs implement this protocol. It is beneficial for IPs that require user configuration and are used to transfer large amounts of data. IP cores that exchange information with each other are connected using AXI master and AXI slave interfaces. Furthermore, the AXI protocol offers

various infrastructure IPs (such as the AXI Interconnect and AXI SmartConnect) to route transactions between multiple AXI master or slave interfaces. The AXI4 protocol offers three types of interfaces:

- **AXI-Full:** Used for high-performance memory applications and supports up to 256 burst transactions.
- **AXI4-Lite:** Low-throughput memory-mapped communication used mainly to access registers.
- **AXI4-Stram:** This AXI4 variation, as mentioned before, provides high-speed streaming and supports multiple data streams that use the same set of wires and various stream types.

5.1.3 PS-PL AXI Interfaces

PS and PL communication is achieved through multiple interfaces and signals, allowing the implementation of various user-created hardware designs and functions in the PL that communicate with the processor and the PS side's memory resources. The ZYNQ UltraScale+ MPSoC architecture provides the following types of AMBA AXI ports that connect the processing system with the programmable logic and vice versa:

- Two High Performance (HP) Master AXI interfaces from PS to PL.
- Six HP Slave AXI ports from PL to PS:
  - Four HP AXI ports from PL to PS DDR.
  - Two HP Coherent (HPC) ports from PL to cache coherent interconnect.
- One port from PL to RPU (PS) for low latency access to OCM.
- One AXI interface from RPU in PS to PL for low latency access to PL.
- One AXI interface (ACP port) for I/O coherent access from PL to Cortex-A53 cache memory.
- One AXI interface (ACE Port) for Fully coherent access from PL to Cortex-A53.

The HP Master AXI ports present higher performance and lower latency for transactions from the PS to PL. However data coherency must be maintained through software. The ports can be used by the Full Power-DMA for data movement between the PS-DDR and PL. However, the FP-DMA is not coherent. For transactions from PL to PS we can choose the ACE port for fully hardware coherency and the ACP port (lower performance) for I/O coherency.

5.1.4 AXI DMA/CDMA

The AXI Direct Memory Access (DMA) IP core provides high bandwidth memory access between AXI4 Memory Mapped and AXI4 Stream IP cores. This high-speed module connects Memory-Mapped to Stream (MM2S) Master interfaces with and Stream to Memory-Mapped (S2MM) Slave interfaces. Both channels operate independently, providing automatic burst mapping and queuing of multiple data requests. It consists of two data movers, one for read and one for write operations, that operate independently. Furthermore, the DMA IP core can be configured to work in polling, interrupt, or scatter-gather mode (instead of writing memory addresses to registers, the DMA controller grabs them from a linked list in DDR memory) and is connected through AXI Interconnects (or in recent versions of Vivado through AXI SmartConnect) to other IPs. In simple mode (polling/interrupt), the processor configures the DMA, and the transaction starts by writing data at the DMA's registers. As parameters, it accepts the source and destination address and the access pattern and, when configured in interrupt mode, sends an interrupt at the processor that the transaction is complete. The integrated Scatter-Gather (SG) engine coordinates data transfers between AXI IPs and the DDR4. The SG operation mode provides the user with the ability to store/request headers from a location in memory and then store/request payload from a different memory location, improving the system's throughput.The maximum memory bus that DMA can support is 1024 Bits in one cycle.

On the other hand, the AXI Central Direct Memory Access (CDMA) IP core provides high-bandwidth Direct Memory Access (DMA) between a memory-mapped (MM) source address and a memory-mapped destination address using the AXI4 protocol. Its function does not differ from the DMA, but it is specially designed for MM to MM transactions.

5.1.5 The Global Address Space

As shown in figure 5.2, the global address space stretches over 1TB, suitable for both 32-bit and 64-bit processors.

The address space's design allows 32-bit processors to communicate with the majority of the board's memories, on-chip peripherals, and other elements. The 36-bit address space allows 64-bit master elements to optimize their access to shared resources such as the DDR or the PL. Finally, 40-bit addresses provide access to the board's processing resources.

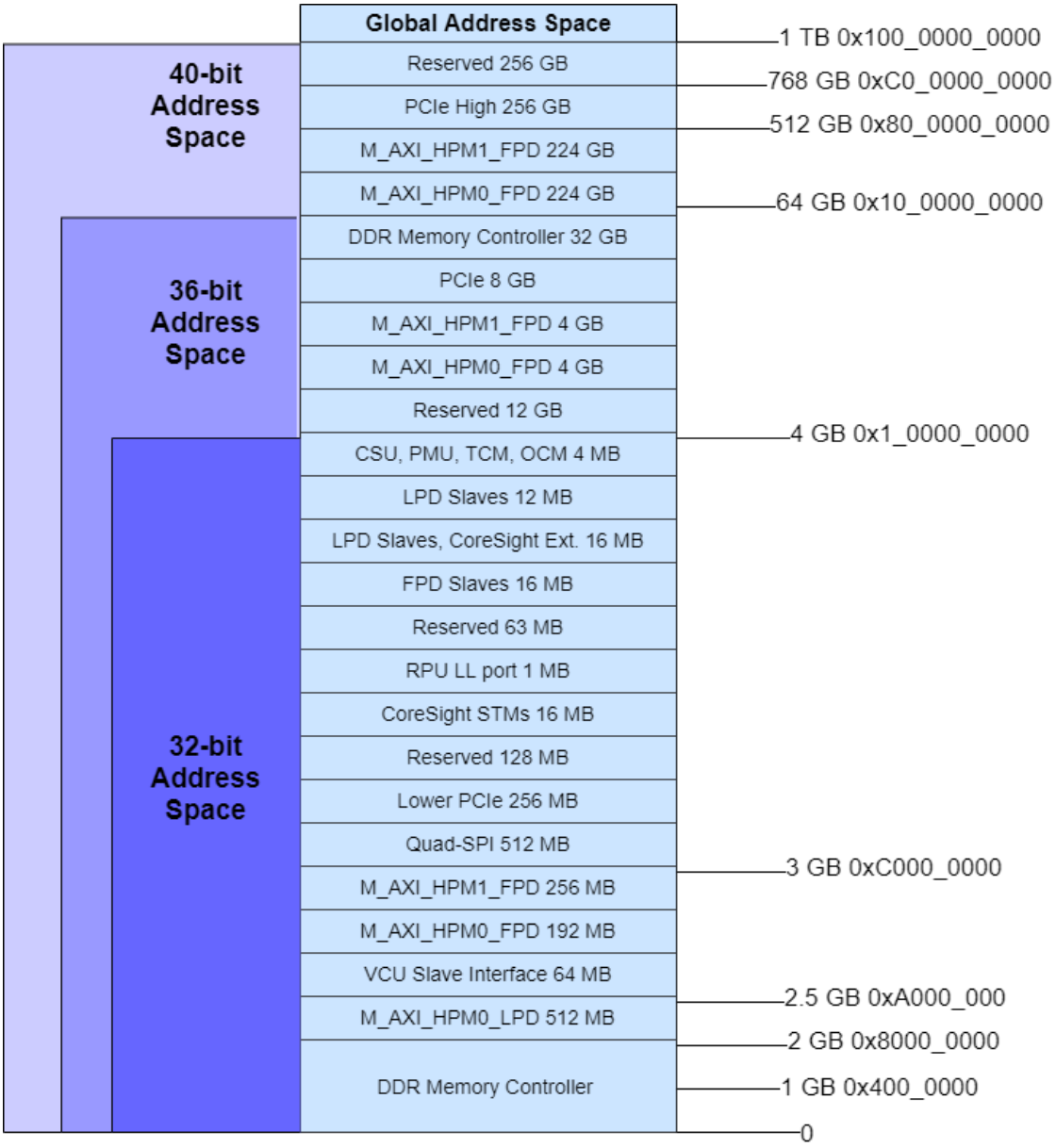


FIGURE 5.2: The ZYNQ UltraScale+ MPSoC global address space.

5.1.6 PS-Side: DDR4 SODIMM Socket

This memory module can achieve nearly twice the bandwidth since it transfers data on both the rising and falling edges of the clock signal (also called and double pumping). Assuming that transferring data are 64 bits wide (which is the most common transfer size nowadays), the DDR's transfer rate (in bytes/s) is estimated as follows:

$$Transfer\ Rate\ (B/s) = \frac{Memory\ Bus\ Clock\ Rate \times 2 \times 64}{8} \tag{5.1}$$

A typical DDR4 memory structure is listed below (note that the dimensions and architecture correspond to the memory module we are using[44]):

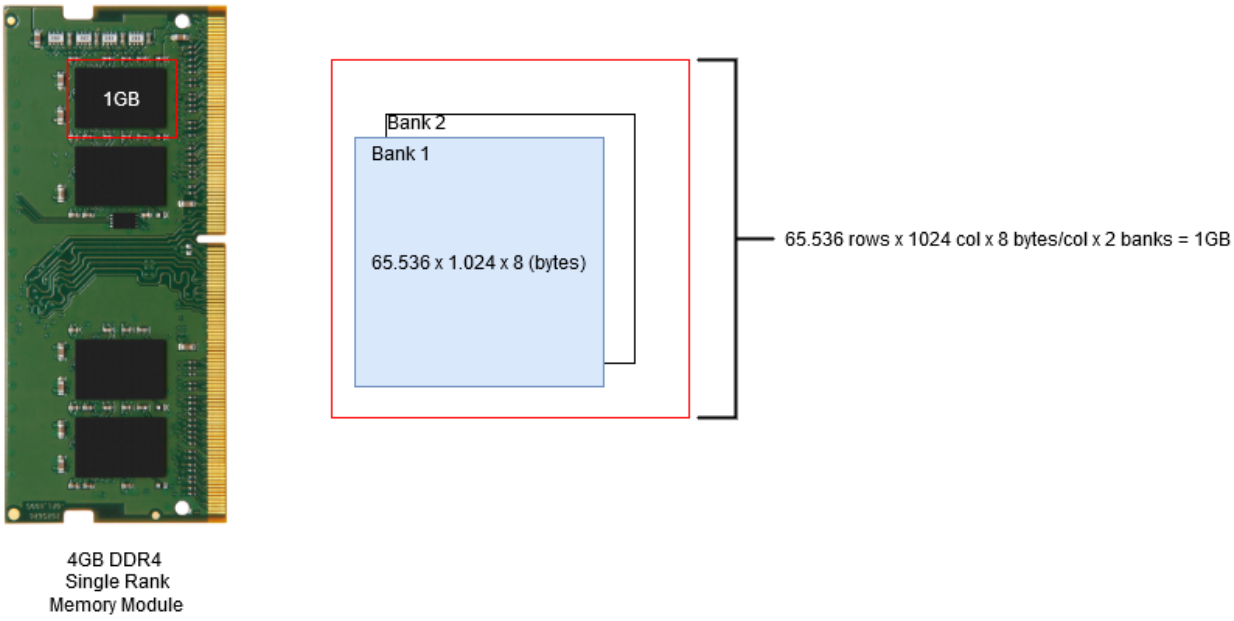


FIGURE 5.3: Memory Architecture of a DDR module.

5.1.7 DDR controller and physical layer

The user interface sends burst transactions to the controller, which in turn generates transactions to/from the SDRAM, taking into consideration timing configurations and refresh actions. Read and write operations are combined in order to reduce dead cycles. The DDR controller, which operates with a system clock to DRAM ratio of 1:4, is also responsible for reordering the commands to improve SDRAM's data bus utilization.The memory controller implements an aggressive precharge policy where it scans the queue of requests as each transaction completes. In case there are no requests for the currently open bank/row, the controller closes it to minimize latency.

The physical layer equips the SDRAM with a high-speed interface. This component includes all the hard blocks (data capture, transmission, serialization, high-speed clock generation) and the soft blocks (memory initialization and calibration) inside the FPGA. The physical layer takes DRR4 commands (active, read/write, precharge, refresh e.t.c.) as inputs and issues them directly to the DRAM bus.

5.1.8 PL-Side: AXI BRAM Controller

This IP [45] provides an AXI4 (memory mapped) slave interface for low latency control of the BRAM resources. Through dual-port FPGA BRAM technology, it provides separate read and write channel interfaces. This core supports 256 beats for INCR bursts and 16 beats for FIXED and WRAP bursts. In INCR bursts the next address is incremented by the data size and is used for a normal memory device while FIXED bursts are used for an address fixed I/O port to make

continual access. WRAP bursts are suitable for cache aligned accesses. It has two BRAM ports that can connect to the same or different Block Memory generators. By default, Port A is designed as the write port and port B as the read port. This means that with the same controller, we can read and write to different BRAM addresses, providing us with more flexibility (i.e., the first port is writing to a separate block generator than the second one, which reads from a different module). In case only one BRAM port is utilized, it can be used for both read and write operations. The most notable drawback of this module is that read/write collisions are not detected to minimize memory resource utilization further. In single-port BRAM utilization, all read and write operations are performed on the read and write address channels, and only one operation is supported per cycle, giving high priority to the read operations. It is clear that with two BRAM interfaces, we can double the bandwidth of the write operation. This comes with a cost; the BRAM controller has to be instantiated on the design, meaning that memory resource utilization increases. The BRAM controller is connected with the ZYNQ processing system using one of the HP master AXI interfaces. By default, Vivado will create an AXI SmartConnect [46] module between the ZYNQ and the BRAM controller when the connection is automatic. This SmartConnect module can be neglected by hand connecting the BRAM controller with the ZYNQ.

5.1.9 PL-Side:Block Memory Generator

The Xilinx Block Memory Generator (BMG) [47] core is an advanced memory module that generates area and performance-optimized memories using embedded block RAM resources in Xilinx FPGAs. Both ports have a read/write interface, are entirely independent, and access shared memory space. Through this module, the BRAM can be configured as a single-port RAM/ROM or true dual-port RAM/ROM. The true dual-port RAM is identical for multi-processor storage applications, while the single-port RAM is best suitable for LUTs. In order to achieve maximum throughput, the write port must not receive single write bursts. Furthermore, each port can be assigned a different operating mode :

- **Write First:** In this mode, input data is simultaneously written into memory and driven on the data output. We chose to use this method since collisions might occur when a read/write operation targets the same port.
- **Read First:** Input data are stored in memory while previously-stored data appear on the output. This method guarantees no collisions, and read operation will access already stored data with safety. However, when this mode is enabled, BRAM consumes more power.

- **No Change:** This is the lowest power consumption method, but collisions are possible when both ports access the same address simultaneously.

In the UltraScale+ Architecture a Block RAM stores up to 36 Kbits of data and can be configured as either two independent 18 Kb RAMs or one 36 Kb RAM [48] . Each block RAM has two write and two read ports (data can be written to or read from either two ports). A 36 Kb block RAM can be configured with independent port widths for each of those ports as 32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, or 1K x 36 (when used as true dual-port memory). A configuration of 512 x 72 port width bits can be implemented when only one read and write port is enabled. Both read and write ports are independent, sharing only the stored data. Each port accesses the same set of memory cells using an addressing scheme. The following formulas determine the physical RAM locations addressed for a particular width:

$$END = ((ADDR + 1) \times Width) - 1 \tag{5.2}$$

$$START = ADDR \times Width \tag{5.3}$$

5.1.10 Memory Subsystems

There are three memory subsystems [13] in the ZYNQ MPSoC architecture. The first level consists of the Cortex A-53 cores, which contain two level 1 cache, one for instructions and one for data storing, a Translation Lookaside Buffer, which contains the most recent memory translations, the Instruction Fetch Unit, and the Data Processing Unit. The second level consists of 1 MB of level 2 cache and the Snoop Control Unit (SCU) responsible for cache coherence maintenance. The SCU is connected with the L1 data cache, has information about the data that exist in the cache lines and copies the data form the cache of one core to another. Finally, the level 3 memory subsystem contains the DDR memory controller, the PL memory modules, and the of-chip DDR4 memory. The Cache Coherent Interconnect (CCI) of the MPSoC together with the AXI interconnect allow hardware coherency to be achieved. The CCI is responsible for data maintenance between cache, cores, PL and, DDR.

For simplicity reasons, in the next sections, the DDR4 memory module and the controller will be presented as one. The APU and its peripherals will contain only the four A-53 Cortex cores.

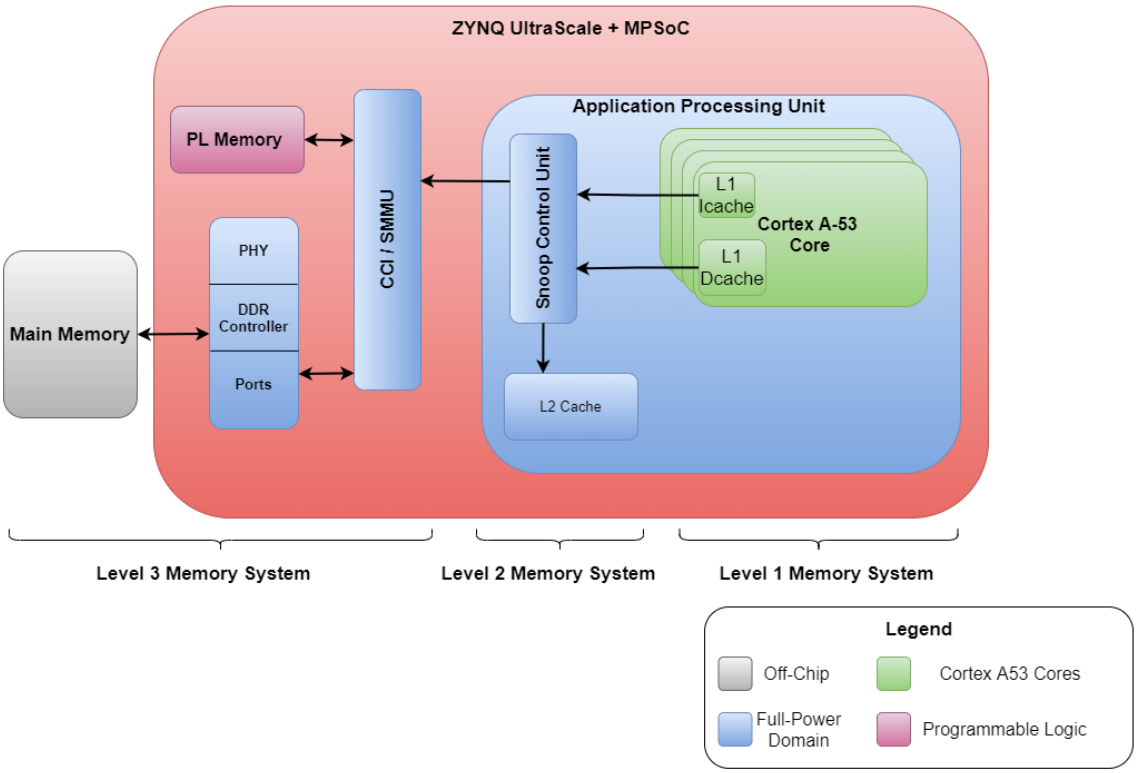


FIGURE 5.4: Memory Subsystems in ZYNQ UltraScale+ Architecture.

5.2 Data Allocation

CNNs require a large amount of data for processing (images, videos - image frames, etc.) and information regarding the network's initialization (weights, parameters, labels) to be stored and used frequently throughout the computation process. FPGAs often come with some GBs of DDR memory, storing data in the PS side throughout the computation period. We can use an SD card to load the parameters (through the ARM cores) on the PS DDR4 memory module. However, SATA hard drives or other storage devices can connect to the device using the SATA or the USB2/3 port. The ethernet or the JTAG port can also be utilized to feed data to the PS DDR4 memory component. Even though SD cards have limited bandwidth, transferring data to the primary storage component is a one time task, happening after the device's boot-up. Since the initialization parameters fit at the DDR4 module, leaving enough space, it is a common technique to load the input data as well. This saves enough time since all the necessary data for the hardware accelerators to perform are located on the DDR on the PS side (which has higher bandwidth than SD cards) and are fed to the rest of the platform using high bandwidth interconnects. We measured that it takes 20.12 sec to transfer 248 MB of parameters from the SD card to the DDR4 memory, an amount of time that can

easily be ignored. However, the SD card would not be an option when we employ CNNs with larger data sets that require more storage space. Moreover, SD cards are useless when hardware accelerators on FPGA systems are used remotely since the files on the SD card have to be frequently updated.

5.3 Structure of the Experiments

The FPGA platform described in section 5.1 has been/is being used in many applications, either in single or multi-FPGA systems. Vivado HLS tool, which is used to implement the hardware accelerators, provides the designer with estimations about each function's latency and clock cycles. The information provided to the designer by such tools alongside the figures presented in this thesis can be used towards FPGA buffer sizing and decisions concerning pipelining parts of the design. Depending on the latency and memory performance results, the designer can decide whether pipelining parts of the design is efficient. To be more specific, depending on the access pattern, it might be more beneficial to request all the parameters concerning a small layer with a single transaction instead of having more scattered accesses in memory.

In order to simulate memory access patterns, we evaluated how memory behaves in individual accesses first. Algorithm 1 presented in Section 5.4.2 calculates the average clock cycles required to read/write a value in specific memory locations. Depending on the byte difference between these locations, a different part of the address bits is changed. However, this experiment does not exploit the burst abilities of the DDR4 memory module. This experiment has three forms:

- At first, each memory location is read 500 times (data read are of no importance) without any writes in memory to test how the memory controller behaves in individual read-only requests which is a common characteristic of CNNs.
- The second form of this experiment includes 500 writes followed by 500 reads of these locations.
- The final form of this experiment includes 500 writes in a location followed by 500 reads of the same memory address, then 500 writes of the second location followed by 500 reads of the second location, etc.

Memory access patterns vary depending on the structure of each application. In 1-D CNNs, such as the one presented in [5], data are stored in consecutive memory addresses. Algorithm 2 presented in Section 5.7.1 simulates the memory access pattern of such applications with data stored and accessed sequentially. Depending on each application's size, we can have multiple Processing Elements (PEs)

requesting data without implementing strides in memory. For these applications, each row can be conveniently mapped to the DDR row and benefit from burst transfers. To evaluate memory's performance in such access patterns and applications, we utilized two PEs that continuously request various data sizes, simulating different layer sizes and applications.

Algorithm 3 presented in Section 5.7.2 simulates the memory access patterns of multi-dimensional CNNs such as the ones presented in Section 3.2 and [49]. These applications require data that reside in non-consecutive memory locations meaning that striding in memory is necessary and only one of the dimensions can be mapped to the DDR row. We implemented an algorithm that sends multiple requests in memory, selecting different stride and transfer sizes for this experiment set. Transfer size varies from several bytes to MB, simulating even bigger layers while keeping in mind the limitations set to us by the hardware resources. Since on-chip memory resources are limited, such requests can not exceed the threshold of 2 MB per transaction, especially when pipelining strategies are implemented. In every iteration of this algorithm, the first PE is assigned with an address and the request size, transferring the required parameters to the corresponding BRAM locations. The second PE is assigned its request size (which can be the same or different as the previous size) starting from a different memory location which corresponds to the stride size. This location can be before or after the first PE's starting address, but both transactions do not overlap. This is done to simulate pipelined architectures where different layers compute simultaneously, and data are needed to be transferred to multiple hardware accelerators.

The results presented in Chapter 6 can be used towards BRAM buffer sizing and pipelining techniques. However, utilizing a different platform mandates an evaluation of the new memory subsystem. The benchmarks presented in this study can guide these evaluations. Characteristics such as the AXI bus width, the DDR4 clock speed, and the memory controller can affect and increase/decrease the performance of such a system.

5.4 DDR4 Evaluation

In this section, we are going to describe all the tests and evaluation techniques we used to obtain as much information we could on how the ZYNQ UltraScale+ MPSoC DDR4 memory subsystem behaves. Contrary to other published researches in the same field [50], [51] which is mostly concerned with DRAM evaluation and effects on memory, we test the DDR itself, observing its behavior on random and sequential accesses, connecting different IPs, and measuring data transfer time.

5.4.1 DDR4 Mapping

In order to proceed to test our memory, we had to study in detail how memory mapping is achieved in the DDR4 module and the UltraScale+ architecture. As we described in section 5.1.6, the DDR4 SODIMM module is internally organized in ranks, bank groups, banks, row, and columns. To access a row and a column and consequently the data within, we have to "pay" a penalty for the activate, read, and precharge commands to access our data. Since we are dealing with multidimensional data, we know that the desired information is not always stored in a sequential manner, or our computational modules require data that reside in different rows, we end up accessing different row addresses every few cycles. This is the easiest way to achieve poor performance. However, there is a solution to this problem(also referred to as ping-pong scheme). By storing data to different banks and bank groups and accessing different banks and bank groups instead of different rows, we are able to hide these protocol hits by masking them with bank switching. So the DDR4 controller can be opening or closing a row in one bank while accessing another bank. In this case, the column address does not matter.

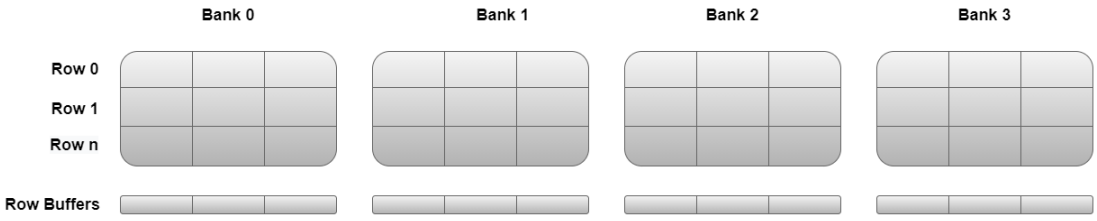


FIGURE 5.5: Data allocation in a DDR4 memory module.

The architecture that we evaluated provides two memory-mapped options on how to access data. Keep in mind that when an address is sent to the DDR from the controller, the bits' position reveals the row, column, bank, and bank group (and rank if the SODIMM module is double-sided). The system uses a two-level interleaving method, where the eight banks are separated into two bank groups, of four banks each. Firstly, the controller chooses the bank group and then the appropriate bank. Even though the memory module has four chips (each chip contains two banks and two chips form a bank group), there are no extra bits to choose between banks in a bank group that resides in different chips. This means that when a bank group is selected, the desirable bank can be found on either two chips that form the bank group.

For the BANK ROW COL memory mapping scheme, the two bank bits are the MSBs, followed by the two bank group bits, followed by 16 row address bits, and finally, the ten column address bits as presented in figure 5.6. This scheme keeps

the bank group and bank address bits as the MSBs, meaning that data are allocated more in rows and columns rather than in banks and bank groups. The CAS/RAS timings are the biggest percentage of the total access time, since we always pay for active and precharge commands when reading and writing sequentially.

SDRAM BUS	Bank[1:0]	Bank Group[1:0]	Row[15:0]	Column[9:3]	Column[2:0]
Address Bits	29,28	27,26	25 through 10	9 through 3	2,1,0

FIGURE 5.6: Address Bits for the BANK ROW COL memory-mapping scheme.

On the other hand, the ROW BANK COL memory mapping scheme is almost the same as the previous one, but the row and the bank group and bank address bits are switched. This method is efficient when most of the requested data target column addressing bits, meaning that the requested data reside in different columns. However, this is not always the case since, to fully utilize the memory bus, CAS delays of the module have to be minimum. In general, to achieve maximum performance and bus utilization, we have to be aware of our application's address sequence and add logic to map our address bits with the highest toggle rate to the LSBs of the memory address.

SDRAM BUS	Row[15:0]	Bank[1:0]	Bank Group[1:0]	Column[9:3]	Column[2:0]
Address Bits	29 through 14	13,12	11,10	9 through 3	2,1,0

FIGURE 5.7: Address Bits for the ROW BANK COL memory-mapping scheme.

The most efficient memory mapping scheme is the ROW COL BANK scheme, where row bits occupy the MSBs and the bank and bank group occupy the least significant bits. This particular scheme is ideal for applications that require data that reside in addresses that increment linearly by a constant step size of hex 8 for long periods. This ensures that the transactions are evenly interleaved across the controller, and the controller resources are optimally utilized. Note that the three LSBs that belong to column address bits (as depicted in the previous images) correspond to SDRAM burst ordering, and since it is not supported by the controller they are ignored.

It is worth mentioning that in Vivado IDE there is an extra option, in the properties panel of the ZYNQ IP which sets the bank group bits as the two LSBs of the memory address bits. By default this is not enabled and it is highly recommended for applications that want to minimize latency to set this particular setting to 1 (allowing the address registers to map the bank group bits to the LSBs of the memory address bits.

5.4.2 DDR4 response time for individual accesses

Our first step was to evaluate the DRR module on the PS side without utilizing the PL memory resources at all. For this purpose, we instantiated only the ZYNQ UltraScale+ module in Vivado IDE, synthesized, and implemented the design as described in Chapter 4. In Vivado IDE, we are allowed to customize the DDR module according to our needs. Throughout the rest of our experiments, we choose to keep the DDR controller enabled in order to achieve maximum performance. It is worth mentioning that Vivado IDE provides us with the expected CAS, RAS to CAS, and Precharge time, which is valuable since we can compare our experimental results with the theoretical ones.

Knowing how the DDR4 is mapped, we wanted to test the two memory mapping schemes using a simple data access pattern. For this purpose we allocated data in different parts of the DDR4 module and accessed them 500 times in order to obtain more reliable results. Every time, we changed a different part of the memory address and kept the rest of the bits. Every word in the UltraScale+ architecture is 8 bytes (64 bits).

For this group of experiments, we randomly selected a basic address in memory where we wrote and read the first number 500 times (this number was chosen randomly as it gives us confidence that the measurements we take correspond to reality). We then measured the write/read time of these individual addresses in clock cycles, and we kept the average measured time for each access. The addressing was based on the ROW BANK COL memory mapping scheme. For these experiments, we chose to increase the memory address by several bytes in each measurement so that only one part of the address changes each time (row, column, bank, and bank group). It is noted that because the memory module of our system consists of only one rank and a DIMM, we know in advance that different memory buses will not be needed. To be more specific:

- 8 bytes distance: If the addresses are 8 bytes the data are in consecutive address spaces, in the same bank group, bank and row.
- 64 bytes distance: If the addresses are up to 64 bytes apart then the data are stored in different columns but in the same bank group, bank and row.
- 128 bytes distance:If the addresses are up to 128 bytes apart the the data reside in different columns, and bank groups but in the same bank and, row.
- 256 bytes distance: The data stored with 256 bytes between them reside in the different column, bank group, and bank but they are still in the same row.

- 2048 bytes distance: Data that are 2048 bytes apart reside in different columns, bank groups, banks, and rows.

It is worth mentioning that the board presets that we have applied to our DDR4 module are 15 bits for row addressing, 2 bits for bank addressing, 1 bit for bank group addressing, and 10 bits for column addressing. These are predefined and are related to the DDR clock that we choose. The DDR module in ZCU102 can achieve a theoretical maximum frequency of 2666 MHz (with a 1333 MHz clock). However, in Vivado IDE, we have tried to clock this device in 2666 MHz, but Vivado would not allow it. So we managed to achieve 2400 MHz frequency with a 1200 MHz clock. That is the maximum achievable DDR clock that is used for the rest of our experiments.

The algorithm below is a simplified representation of our access pattern. To ensure that we read and write in the DDR4 memory, we looked at the global addressing space map and chose an address that corresponds to the DDR4 controller. Then we proceed to write and read the same address 500 times to make sure that the reported clock cycles are accurate. Then we changed the address in which data are stored and written in order to evaluate how the DDR4 controller behaves in different scenarios. We noticed that when we disabled the cache, both read and write operations doubled in latency as was expected. The cache can be useful when enabled for short bursts of data immediately required to be sent to the PL. We also made sure that read/write timings are independent; we wrote at a specific address and read it, or we wrote at all the addresses and read them at the end, and we even read unwritten addresses (their content was irrelevant). The results were the same in all the cases.

Algorithm 1 Measure time for individual access

```
1: procedure SET ADDRESS
2:     write_addr ← base_addr
3:     read_addr ← base_addr
4:     Calculate_Time(write_addr, read_addr)

5:     write_addr ← base_addr + 8 bytes                ▷ Choose the next address
6:     read_addr ← write_addr
7:     Calculate_Time(write_addr, read_addr)

8:     write_addr ← base_addr + 64 bytes               ▷ Choose different column
9:     read_addr ← write_addr
10:    Calculate_Time(write_addr, read_addr)

11:    write_addr ← base_addr + 128 bytes              ▷ Choose different column and
   bank group
12:    read_addr ← write_addr
13:    Calculate_Time(write_addr, read_addr)

14:    write_addr ← base_addr + 256 bytes              ▷ Choose different column, bank
   group and bank
15:    read_addr ← write_addr
16:    Calculate_Time(write_addr, read_addr)

17:    write_addr ← base_addr + 1024 bytes             ▷ Choose different column, bank
   group, bank and row
18:    read_addr ← write_addr
19:    Calculate_Time(write_addr, read_addr)
20: procedure CALCULATE_TIME(write_addr, read_addr )
21:    for i:=1 to 500 do
22:        t1 ← start_time
23:        write_data_to_mem(write_addr, data)
24:        t2 ← stop_time
25:        total_time_w ← t2 − t1
26:    for i:=1 to 500 do
27:        t1 ← start_time
28:        data ← read_data_from_memory(read_addr)
29:        t2 ← stop_time
30:        total_time_r ← t2 − t1
31:    return total_time_r, total_time_w
```

Furthermore, Vivado IDE provides us with the opportunity to configure the DDR4 controller address map as BANK ROW COL. We tested this configuration, but this time the address "jumps" were higher, since to change bank group we had to move 4.2 MB of addresses. This did not affect the performance of the DDR4 memory module and execution of read/write operations resulted in similar number of cycles, which corresponds with recent publications [50]. Read latency is not affected and changing the memory address map is useful only in very small bursts. If a memory module is organized in bank groups then a decrease in throughput is expected when we choose the BANK ROW COL mapping scheme. Taking into consideration the fact that address mapping does not seem to have positive/negative results on our performance for individual accesses and that CNN's memory access patterns require row bits at the MSBs of the address to achieve better performance, we choose the ROW BANK COL memory address mapping scheme (which is the default in Vivado IDE) for the rest of our experiments.

5.5 Memory Management Attributes

The ordering of access in a memory space is defined by the memory attributes which can be proven useful depending on how we are planning to use the memory locations. To further speed the transactions, we used memory attributes that enabled us to use cache for the DDR and the BRAM. The Memory Management Unit (MMU) can be enabled/disabled as wished, and further configurations regarding the memory can be applied.

5.5.1 Normal Memory

This attribute applies to most memory systems, which allow unaligned accesses. This memory type is appropriate for locations where multiple read access returns the last value stored in these locations or when we can issue repeated write access if the contents of the location do not change between the write operations. This memory type can be Inner (innermost caches)/Outer(outermost caches) shareable or non-shareable. Designers can use this attribute to specify locations of memory where coherency is crucial. If a Normal memory is marked as non-shareable, it can be accessed by one processing element. Besides shareability, normal memory locations are also cacheable. The normal memory attribute can be configured alongside write-through cacheable, write-back cacheable, and non - cacheable. In write-through, data are simultaneously written to the cache and memory, and this method is best used when data are not frequently written to the cache. The data are updated only in the cache in the write-back mode. The data are written in main memory only when the cache line is ready to be replaced. These attributes provide

coherency controls and using the write-through, or non-cacheable attributes is better for controlling coherency than using hardware coherency methods. If a piece of memory is not going to be accessed shortly after, then a non-cacheable attribute must be enabled.

5.5.2 Device Memory

This type of memory attribute is ideal for MM peripherals where the access to a location returns values that depend on the number of accesses performed before. There are three types of device memory; gathering (G), reordering (R) and, early write acknowledgment(EWA). A memory location that corresponds to peripherals is ideal for write acknowledgment. When a memory location performs gathering, it also performs reordering and EWA. This memory type is Outer shareable, meaning that accesses are coherent and all accesses are aligned. Furthermore, there is no cacheability in this type of memory; cache can not hold a memory location that is configured with this attribute. Gathering allows multiple writes/reads (as long it is only writes or only reads) to the same location to be merged in a single transaction. Reading data from a non-Gathering memory region does not come from cache or buffers; it refers directly to physical locations.

5.5.3 Cacheable VS Shareable Memory

The shareable attribute is used to define whether a location is shared with multiple cores. By default (looking at the translation table files in Vivado SDK) the DDR is mapped as Normal Write Back Cacheable memory while PL is initiated as strongly ordered memory (shareable but not cacheable). This means that the DDR is shareable (whether parts of it are shared by one or all cores can be defined by specific attributes) and cacheable meaning that data are stored in the cache before accessed by other elements. The PL BRAM is outer shareable, meaning that all devices can perform read/write operations.

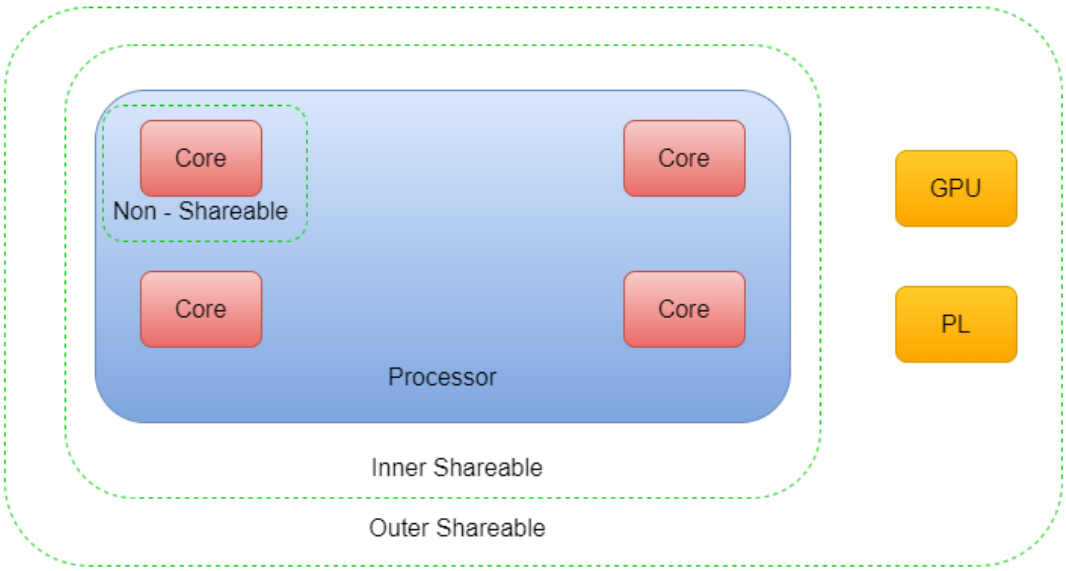


FIGURE 5.8: Inner and Outer shareability.

5.6 Block RAM in Cascade Mode

Since today's computationally demanding applications require a considerable amount of data fast, deep memory capabilities must be exploited. This is achieved by cascading multiple BRAM modules. In order to do so, the requested BRAM must be less than 72 bits in width and more than 512 bits in height. In that case, the BRAM instances will be automatically placed in the same block RAM column. In the UltraScale architecture, the output data of one RAMB36E2 module will be connected to the input data of the next RAMB36E2 module, creating a deeper block of RAM. Data flows from the lower BRAM to the upper BRAM, and all the block RAM modules are connected to the same clock source. The cascade mode offers great performance and power optimization.

In the image 5.9, we can see how the cascade mode is implemented in hardware using multiplexers and registers. This register is utilized only when we chose to operate the BRAM in pipelined mode. Data from previous BRAMs are connected to the first multiplexer of the next BRAM to be pipelined in the current stage or to the final multiplexer of the current BRAM module, which decides the output; either the current BRAM's data, the cascaded data from the previous BRAM or the data from the current BRAM module before the register (this particular action is also known as fall-through, were the data of a memory module are forwarded to the output before written to the pipelined register to gain one clock cycle). There are three types of cascade mode available:

- **Standard Data Output:** In this type of cascade the data from the lower BRAM are only connected to the final multiplexer of the current BRAM

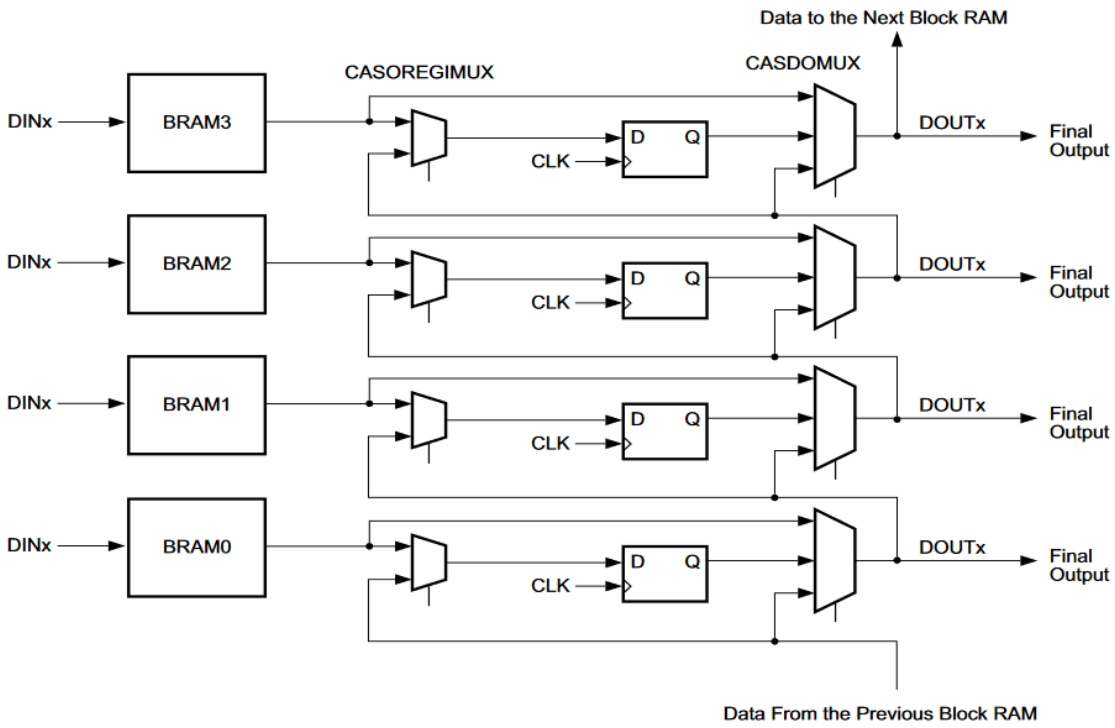


FIGURE 5.9: The BRAM cascade architecture. Copyrights to: [48].

(and not in the first one) and can be applied to the entire column creating a very deep memory structure with only a few logic resources.

- **Pipelined Data Output:** In this method the cascaded data are connected to the first multiplexer (before the optional register) and there is no fall-through. This means that the output data are always pipelined whether they are comming from the lower or the current BRAM module.

- **BRAM Array Matrix in Systolic Mode:** In the final type of the cascade mode, the cascaded data are connected to the first multiplexer (before the optional register). Cascaded data are always pipelined (meaning that they go through the register) whereas the data from the current BRAM are not; they can either be pipelined or fall-through to the output.

BRAM can be accessed by the Block Memory Generator (BMG) or by inferring BRAM in HDL code; we chose the first option. BMG is connected with a BRAM Controller (BC), which allows the BRAM interface to be connected with the ZYNQ AXI ports. Since BMG acts as a slave in this setup and the BC as the master, all the BC configurations are propagated to the BMG. In order to change configurations, as a workaround, we can set the BMG's mode into standalone, configure the width and depth of our BRAM and change back to the BRAM controller mode. In this way, we managed to create larger BRAMs in depth. The default algorithm used to concatenate the block ram primitives is the Minimum Area algorithm, providing an optimized design with a minimum number of BRAM

primitives and reduced output multiplexing. Vivado IDE offers the opportunity to change the cascade height of the RAMB36E2 BRAM modules used in a design. By default, if the user takes no action, Vivado will decide the cascade height.

When expanding the schematic created by our design, we noticed a slight difference between what is depicted in [48] and in this video [2]. The latter represents a slightly altered version of the BRAM cascade scheme where the cascaded output from the lower BRAMs is also written in the upper BRAMs, besides being passed in the multiplexers after the BRAM module. Our schematic is identical with the one presented in 5.9. Due to lack of information on the subject, we posted in Xilinx's forums [3] to clarify the matter. It appears that in order to gain direct control over cascading (meaning that we decide the cascade height), we have to discard the BMG and, in its place, instantiate an XPM memory. XPM memories are part of Vivado's XPM libraries[52], which can be instantiated in our design by selecting the appropriate XPM template. In contrast to BMG, a pre-defined IP core, XPM libraries are parameterizable and allow the user to create larger cascaded BRAM blocks. Vivado already offers an optimized solution with the BMG, so if the application requires data cascading, the XPM memory template must be used instead of BMG.

5.7 Memory response time for burst data accesses

We then expanded our study to see how DDR4 responds when mass amounts of data are requested. Up to this point we did not utilize the PL side at all to request data and we kept everything in a software level. In order to test how memory responds to burst data accessing we requested multiple KB or MB of data that reside in sequential or random rows. The ARM CPU cache line has a size of 64 bytes in the UltraScale+ architecture, and since the ARM also controls the DDR4 controller, the burst size of any transaction should be a multiple of 64 bytes in order to achieve maximum performance from the controller.

To test all those cases we utilized the BRAM controller (BC) and the Block Memory Generator (BMG). The BMG's interface corresponds to a typical BRAM module and has no AXI ports to connect directly with the ZYNQ PS. Considering this, the BC is responsible for receiving information through its AXI ports and sending the information through its BRAM interface connected with the BMG. We connected the two modules together and then with the PS using the two master AXI ports, without any interconnects as intermediary (to avoid any unnecessary

[2] https://www.xilinx.com/video/fpga/optimize-ultrascale-architecture-block-rams-low-power-high-pe
html

[3] https://forums.xilinx.com/t5/Xilinx-IP-Catalog/BRAM-in-cascade-mode/td-p/
1199261#M9276

multiplexing in PL that could slow the design). We used both ports of the BMG, and configured it so both port A and B are in Write First mode (Vivado IDE does not provide the opportunity to configure the BMG ports in other modes when we connect it with the BC, it has to be in standalone mode in order to configure the ports which was not our case).

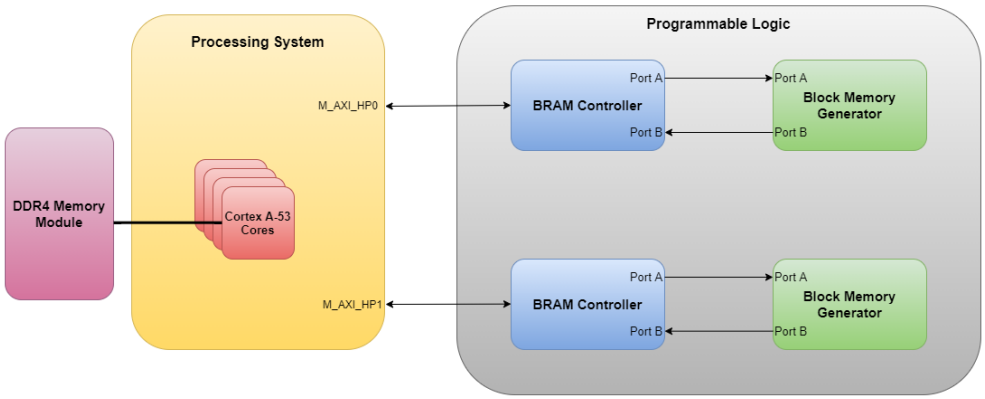


FIGURE 5.10: Block diagram of the experiment.

Each BMG module occupies 1MB of BRAM space. The available BRAM is 4MB, so we chose to utilize 2 MB of BRAM to test our cases. Our DDR4 module was clocked at 2400 MHz DDR frequency, which means we had a 1200 MHz clock. The DDR to PL clock ratio is 1:4, so the PL fabrick clock was set to 300 MHz to achieve the maximum throughput. The DDR module has 64 pins, which means that at 2400 MHz, it can provide a theoretical peak throughput of 19.2 GB/s.

At first we repeated the previous test described in 5.4.2 case where we chose random addresses that reside in different columns, banks, bank groups and rows and transferred data from the DDR to the BRAM using the two Memory Mapped (MM) modules. Then we tested how memory behaves in burst transactions, writing data from the PS to the two BMGs and vice-versa. We ensure that we access different rows, columns, banks, and bank groups with different access times to evaluate how DDR4 behaves when data are needed at high speeds. The data are loaded in the cache and then flushed to specific BRAM addresses. It is important to mention that when we transfer data from PS to PL, meaning that PS is writing the data and the PL is reading them, only flush is required in order for data to be written in the physical addresses. However, when data are written by the PL and read from the PS, then a space of *destination address + offset* (where the offset is the number of bytes we transferred) needs to be invalidated. It is vital that when we use the cache, buffers are cache-aligned; in ZYNQ, UltraScale + cache line is 64 bytes, and the buffers need to be aligned with the cache line; otherwise, the

cache might end up contain stale data. If both PS and PL write at the same time then we can not use the cache.

We used some of the memory attributes we mentioned in order to speed up our design. To be more specific, we experimented with data cache disabled and enabled. Additionally, we used the memory attributes to the memory space reserved by the BMGs, using the normal write-through and write-back inner shareable and cacheable attributes of the normal memory type. What is more, the designer must know where the memory accessed resides in the global address space; if the memory is less than 4GB then the attribute is set for a section of 2 MB memory, starting from the base address. Whereas, if the memory is larger than 4 GB then the attribute is set for a section of 1 GB, starting from the base address. If we need to store data in a region that resides more than 2 MB of the base address, then have to configure the memory attributes for that region as well. We need to know that when we place an attribute on a memory section, then we cannot apply a different attribute for 2MB or 1GB after the base address. In case we need to change our access pattern (and subsequently the memory attribute) to exploit its capabilities fully, we must reserve address space that does not reside in the already attributed memory region.

5.7.1 Sequential Burst

For this set of experiments, we initialized the DDR memory with 64-bit double-precision floating-point data. Then we proceeded to sequential bursts, starting from some Bytes to some MB. The DDR4 module that we use has 2 KB of page size (number of bits per row). This means that when a row is activated, 2 KB of data are loaded into the Sense Amps. When accessing data in consecutive memory addresses, the CAS delay plays a vital role since column switching is done much more than row (RAS) switching.

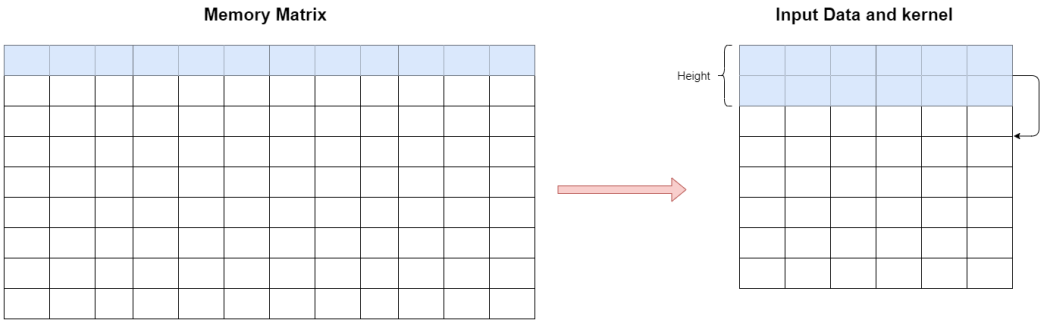


Figure 5.11: 1 - D Convolutional Neural Network and data allocation in memory.

Sequential data access is ideal for 1-D CNNs (depicted in 5.11 where the filter (kernel) moves in one direction (in this example vertical). Data are stored in consecutive addresses and can be obtained with a single burst (depending on the input image's size and the kernel's height). The filters' parameters are located in consecutive memory locations, and the use of a DMA is ideal for maximizing the throughput of the design.

To test the memory response in sequential accesses we transferred data from the PS to the PL using both BMGs. The first BC is assigned an address, the base address of the DDR and the second BC is assigned with the *base address + offset*, where the offset depends on the number of bytes we want to transfer each time. This means that for small bursts (smaller than the page size) both BCs are going to access the same memory row. Then the data are transferred in two different addresses in the PL (since we have two BCs). This means that the hardware accelerator can read data from both adddresses simultaneously.

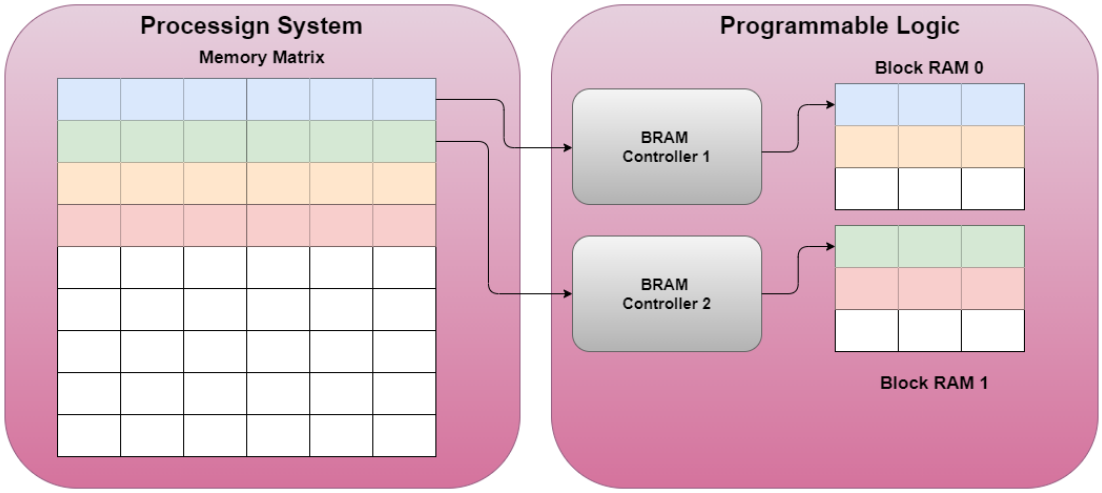


FIGURE 5.12: Sequential access from two BC.

In this experiment, we accessed the memory 100 times, bursting some bytes each time. We started from a *base address* where we sent 640 bytes to BC0, then we sent 640 bytes to BC1 from *base address+640 bytes*. Then BC0 received another 640 bytes from the *base address* + 2×640 *bytes*, e.t.c. Then we proceeded to increase the transfer data size to 2KB (page size), 20 KB, 200 KB, 2 MB for every BC. In order to simulate large data requests (for example a layer in a CNN needs 150 MB of data) we ran iterations. In every iteration each BC transfers 2 MB of data (total 4 MB for an iteration). We requested 50 iterations (totaling 200 MB of sequentially stored data transferred to the PL). These are transfer sizes that represent the memory footprint of various applications, from smaller to larger

ones. Note that since we have only 2 MB of BRAM space available, data in bigger transfer sizes are overwritten.

Algorithm 2 Measure time for sequential burst access from PS to PL

1: **procedure** BURST DATA
2: $Set\_attribute(bmg0\_addr, MEMORY\_ATTRIBUTE)$
3: $Set\_attribute(bmg1\_addr, MEMORY\_ATTRIBUTE)$
4: **for** i=1 **to** number_of_iterations **do**
5: $Bytes \leftarrow set\ number\ of\ bytes$ ▷ this depends on the transfer we do every time
6: $t_1 \leftarrow start\_time$
7: $burst\_data(from\_ddr\_addr,\ to\_bmg0\_addr,\ Bytes)$
8: $flush\_memory(bmg0\_addr,\ Bytes)$
9: $t_2 \leftarrow stop\_time$
10: $get\_results(t_2 - t_1, Bytes)$

11: $Bytes \leftarrow set\ number\ of\ bytes$
12: $t_1 \leftarrow start\_time$
13: $burst\_data(from\_ddr\_addr + Bytes * i, to\_bmg1\_addr, Bytes)$
14: $flush\_memory(bmg1\_addr, Bytes)$
15: $t_2 \leftarrow stop\_time$
16: $get\_results(t_2 - t_1, Bytes)$

17: $ddr\_addr \leftarrow ddr\_addr[Bytes * (i + 1)]$
18: **procedure** GET RESULTS(time, Bytes)
19: $Throughput \leftarrow (Bytes * CPU\_CLock)/(2 * time)$
20: $Latency(us) \leftarrow (time * 1000000)/(Counts\_per\_second)$
21: $cpu\_clocks\_per\_word \leftarrow time/(word)$ ▷ word = Bytes / 4
22: $fpga\_clocks\_per\_word \leftarrow fpga\_clock/(cpu\_clock * cpu\_clocks\_word)$
23: **return** $(Throughput, Latency, cpu\_clocks\_per\_word, fpga\_clocks\_per\_word)$

In the 13th line the second BC requests data that reside after the bytes transferred from the first BC. For example, if the first BC starts from the address *0x0* and transfers 640 Bytes then the second BC will start from the address *0x0 + 640 Bytes* and end at $0x0 + 2 \times 640 Bytes$ address. Note that the time it took to transfer data is multiplied by two in the 19th line since the counter counts every two clock cycles and the measured time provided by the software equals half the actual transfer time.

5.7.2 Random Burst

In this set of experiments memory accesses are not consecutive. Every access contains a stride in addresses that depends on the size of each application. Data can be stored in different banks, bank groups or rows.

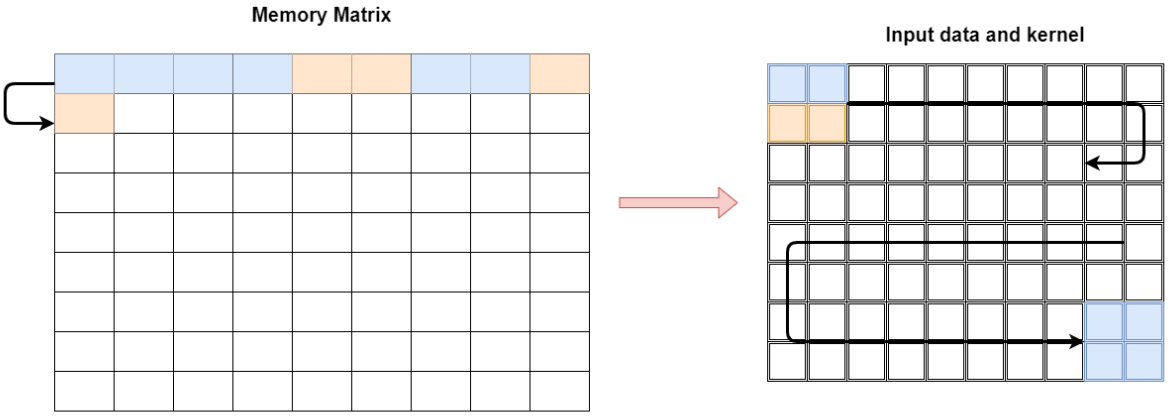


FIGURE 5.13: 2 - D Neural Network and data allocation in memory.

In the 2-D category, we can assume that in most implementations, data reside in random addresses. The architect must decide whether to configure the access pattern as sequential or random. A 2D convolution layer means that the input of the convolution operation is three-dimensional, for example, a color image which has a value for each pixel across three layers: red, blue and green. However, the movement of the filter across the image happens in two dimensions. The filter is run across the image three times, one for each of the three layers. In this section, we requested data that reside in non-consecutive memory addresses. When we have CNNs with multiple filters, and dimensions data reside in addresses that are far from each other. This makes the use of a DMA difficult and slows down the design. We requested data that are far away (more than one row) and tested the memory response. Moving in strides is usual in large CNNs where the input image is filtered in many dimensions (many filters in each layer).

The image 5.14 depicts the data allocation in memory for a multidimensional application where each filter has its own data sequentially stored (a typical case). But in order to pipeline the architecture and optimize it, we have calculate different pixel results in parallel.

In this set of experiments we accessed the memory in strides. We started at the DDR base address where we burst 640 Bytes with the first BC, then move 1KB (less than the page size) to burst another 640 Bytes with the second BC. Depending on the size of the stride the data can be obtained from different banks or bank groups. If the stride is more than 2 KB (page size) then we need to access

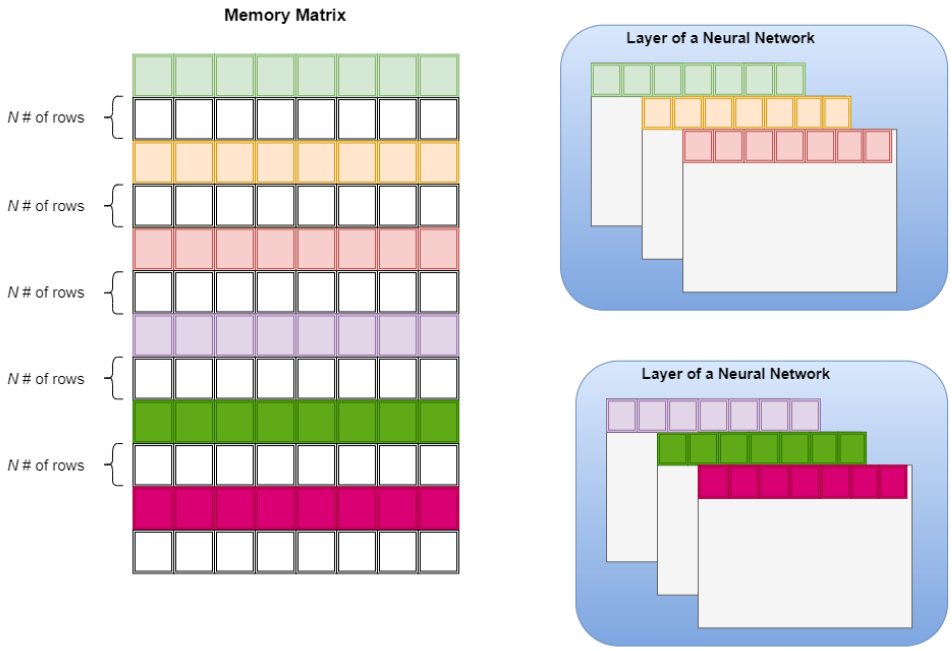


FIGURE 5.14: Multi - dimensional allocation in memory.

a new row. Accessing a new row (RAS) is costly and is the main reason why random access patterns have high latency.

Algorithm 3 Measure time for random burst access from PS to PL

1: **procedure** BURST DATA
2: $Set\_attribute(bmg0\_addr, MEMORY\_ATTRIBUTE)$
3: $Set\_attribute(bmg1\_addr, MEMORY\_ATTRIBUTE)$
4: **for** i=1 **to** number_of_iterations **do**
5: $Bytes \leftarrow set\ number\ of\ bytes$ ▷ this depends on the transfer we do every time
6: $Stride \leftarrow set\ stride\ length$ ▷ This depends on the stride we choose each time
7: $t_1 \leftarrow start\_time$
8: $bust\_data(from\_ddr\_addr,\ to\_bmg0\_addr,\ Bytes)$
9: $flush\_memory(bmg0\_addr,\ Bytes)$
10: $t_2 \leftarrow stop\_time$
11: $get\_results(t2 - t1,\ Bytes)$ ▷ Presented in algorithm 2

12: $Bytes \leftarrow set\ number\ of\ bytes$
13: $Stride \leftarrow set\ stride\ length$
14: $t_1 \leftarrow start\_time$
15: $bust\_data(from\_ddr\_addr + Stride + Bytes, to\_bmg1\_addr, Bytes)$
16: $flush\_memory(bmg1\_addr, Bytes)$
17: $t_2 \leftarrow stop\_time$
18: $get\_results(t2 - t1, Bytes)$

19: $ddr\_addr \leftarrow ddr\_addr([2 * Bytes + Stride] + 1)$

Finally, in figure 5.15 we present an example of how the BCs and BMGs can be used alongside hardware accelerators. The BMG must have both its ports (portA and portB) enabled whereas the BC needs to enable only one (if only one port is enabled then it is a read/write port). The first BC can use its portA (which is a write/read port) to connect to portA of the BMG (each port of the BMG has a read and write channel) and write data. The second port of the BMG can be connected to the read port of a different BC with two ports enabled. Port A of the second BC (the write port) can be connected to the hardware accelerator in order to write the data from the BRAM. In this way, we can use one port of the BC to write the data to BRAM and the other to read the BRAM data and send them to the accelerator.

The purpose of the controller and the BMG is to act as intermediaries so that the hardware accelerators do not have to access the DDR every time they require data. Instead, they can search the BRAM, which the PS will write the data as the computations are calculated simultaneously. The BMG (which instantiates BRAM

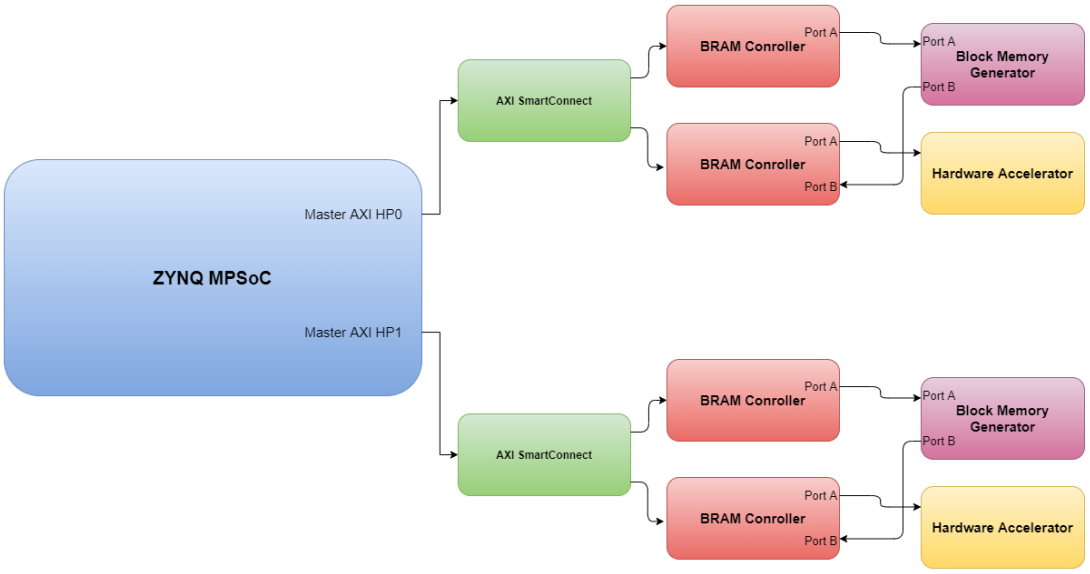


FIGURE 5.15: Example of an architecture that uses BMGs, BC and custom hardware accelerators.

modules that are very fast) can act as an intermediary, where the data required for various computations are stored. That way, by bursting a considerable amount of data to the BMG, the memory accesses from the accelerators are reduced (the accelerators do not have to communicate with the PS at all, only PL to PL transfers are required). In our case, this is difficult to implement due to a lack of BRAM resources. Modern FPGAs have more FPGA memory resources and are ideal for this kind of proposed method. Instantiating BRAM through BMG and using it as a temporary storage unit (combined with faster DDR memory controllers) is worth exploring.

Chapter 6

Results

As it is clear from Chapter 5, our implementation is highly connected with the FPGA platform that we use. A bigger FPGA (meaning more BRAM and many ports from PS/PL to PS/PL) will provide better results and memory bandwidth. Our job was to evaluate the memory subsystem of ZCU102 with Xilinx IPs for various data access patterns utilizing the FPGA to the maximum of its potential while at the same time maintaining low energy consumption and low resource utilization. The work of Georgios Pitsis [36] gave excellent results in one-dimensional data handling in both ZCU102 FPGA and QFDB [4], but this time we are dealing with multidimensional data.

6.1 Results for DDR4 individual accesses

The most essential element of the ZCU102 is the DDR4 memory module, since due to its large capacity, it is ideal for storing many data that can then be fed to the PL part of the FPGA. For this reason, we have looked at this element extensively to know exactly how it can behave when used as a storage element for a computationally demanding network. Note that the DDR4 module is clocked at 1200 MHz for this set of experiments.

Initially, our first test cases involved accessing memory in individual addresses to get a first-hand look at how memory and its controller behave in individual data accessing that reside in different locations. For this reason, we do 500 reads/writes in specific memory locations, each time changing an item from its address (row, column, bank group, bank). Note that we measured the time for read/write with and without cache enabled. The table below refers to the average of the measured clock cycles for each read/write. For this particular set of experiments we did not utilize the PL part. The results presented below utilize only the software part.

Addr. Distance	Base Addr.	Diff. Col. (64 bytes)	Diff. B.G. (128 bytes)	Diff. Bank (256 bytes)	Diff. Row (2048 bytes)
Read /w cache	64	63	63	63	63
Write /w cache	64	64	64	64	64
Read w/o cache	118	118	119	119	119
Write w/o cache	117	118	119	119	119

TABLE 6.1: Average number of cycles for individual read/write operations.

From the table above, we can see that for individual read/write operations the memory controller is efficiently handling the requests we made. The above experiment was implemented many times, and every time we got almost the same results. The variation of the above presented results were minor and they did not need to be presented. Note that since we did not use the PL, all the on-chip memory resources remain non-utilized. The number of clock cycles presented is an estimation; memory controllers present various stalls, which can reduce our measurements' preciseness.

We repeated the experiments when we request 8 bytes (1 word), 500 times from the PS to the PL. Data are stored in different banks, rows, columns and, bank groups. The results are presented below:

Addr. Distance	Base Addr.	Diff. Col. (64 bytes)	Diff. B.G. (128 bytes)	Diff. Bank (256 bytes)	Diff. Row (2048 bytes)
PS → PL /w cache	102	103	103	103	103
PS → PL w/o cache	122	122	124	125	130

TABLE 6.2: Average number of cycles for individual data requests from the PL.

This experiment was conducted multiple times and we noticed that the location of data did not affect the transfer time; which can be confirmed by 6.1; DDR controller is effectively handling the requests. For this experiment we disabled the cache (again) to make sure that we read and write to physical addresses. After analyzing the results (we used an ILA core to examine how data are transferred from the PS to the PL and when the read/write signals are enabled) we noticed a 3 cycle loss per transfer from the BC. This means that the BC can achieve only 25 % of the available throughput. However the BMG can achieve 100 % on both channels (read and write). With no cache involved we would be able to write one

word every 4 cycles. Since the BMG has higher capabilities than the BC in the future we can create a custom full AXI slave controller (such as the one presented here [1]).

The tables 6.1 and 6.2 depict the difference in clock cycles between the DDR to DDR transfers and the DDR to PL transfers. We can understand that in order to write one word from the PS to the PL, we need 102 (on average) clock cycles. The difference that cache makes can be understood when we measured the data transfer time without the cache enabled. The slight increase in transfer time when data are stored in different rows reveals that the DDR memory controller and the cache work together when we request data from a specific memory address and assume that the following addresses will be requested. So in order to increase performance, the row buffers are loaded into the cache lines.

6.2 Results for burst data accesses

In this particular set of experiments we utilized the PL by creating two Block Memory Generators (BMG) and two BRAM Controllers (BC) that configure the BRAM. Our design is clocked at 2400 MHz,(with a 1200 MHz clock) and the PL fabric clock's frequency is set to 300 MHz. Each HP Master AXI port has 128 bit width meaning that the maximum achievable throughput for each port is 4.8 GB/s. Vivado's reports on utilization and power consumption are presented in the tables below.

Resource	**Utilization**	**Available**	**Utilization %**
LUT	2636	274080	0.96
FF	1702	548160	0.31
BRAM	512	912	56.14

TABLE 6.3: Utilization report when using MM Xilinx IPs.

Dynamic (3.45 W ≃ 83%)			**Static (0.73 ≃ 17%)**	
Clocks, Signals, Logic	**BRAM**	**PS**	**PL**	**PS**
0.17 W (≃6%)	0.521 W(15%)	2.764 W(79%)	0.63 W(87%)	0.1 W(13%)

TABLE 6.4: Power report when using MM Xilinx IPs.

Note that the 512 BRAM blocks correspond to 512 RAMB36E2. BMG provides a report at the early stages of the design (before Synthesis and Implementation) that depicts how the BRAM is allocated (either using RAMB36E2 or

[1] https://zipcpu.com/blog/2019/05/29/demoaxi.html

RAMB18E2 modules). Each BMG utilized 256 of such blocks, a number that is a power of 2. As it appears, Vivado will always utilize N BRAMs where N is a power of 2; even it means that more BRAM than requested is utilized. In our case Vivado utilized $512 \times 36Kbit = 2.3MB$ when in fact it should utilize $444 \times 36Kbit = 2MB$ a 15.32% increase in BRAM utilization (approximately 50 KB more memory space than requested). This percentage might appear relatively small, but since BRAM space is extremely scarce, we have to consider the possibility for improvement in memory allocation by the tool. A similar topic was discussed here [2] community forums where if the width of a BRAM was not a power of 2, then Vivado would allocate N BRAM modules, where N is a power of 2, utilizing more memory than requested. Note that for all our experiments, in order to avoid this problem, we set BRAM's width and height as a power of 2.

6.2.1 Results for sequential data accessing

In this set of experiments, we read consecutive data addresses and store the data to the PL. The amount of data acquired by each BC depends on the kernel window's size and the application in general. We observed the throughput of both BCs at various times. We saw that although BC0 does not show large fluctuations, BC1 initially seems to have a larger throughput, which decreases until it stabilizes. When we burst small amounts of data (less than a page size), we noticed that the second BC completed the transaction much faster than the first BC (sometimes BC1's latency was half the latency of the BC0). This happens because data are stored in the cache line, and when the second BC begins, the burst transaction data are loaded from the cache and not the actual DDR memory. However, as the total bytes transferred increased, we noticed that the second BC's throughput dropped until it reached a steady point. We calculated the average throughput and latency of each BC and the total average throughput and latency. We transferred 640 bytes in each transaction (640 bytes for each BC), totaling around 64 KB of sequentially stored data per BC. Then we expanded our experiments and in each transaction we transferred 2 KB of data, meaning that in every iteration of algorithm 2 we transferred 4 KB of sequentially stored data in the BRAMs, totaling (after 100 iterations) 200 KB of transferred data. Then we did the same thing, changing the burst size to 20 KB, 200 KB, 2 MB for every BC. The sequential access from each BC is utilized as it is presented in 5.12.

After the transfer size of 20 KB for every BC (totaling 40 KB of sequentially stored data for every iteration) we noticed that both AXI ports produced similar throughput that did not vary as the total transfer size increased meaning that the

[2] https://forums.xilinx.com/t5/Synthesis/Problems-with-inferring-RAM-with-a-depth-which-is-not-a-td-p/1044294

Transfer Size for each BC	Number of iterations	Total Bytes transferred
640 Bytes	100	64 KB
2 KB	100	200 KB
20 KB	100	2 MB
200 KB	100	20 MB
2 MB	100	200 MB

TABLE 6.5: Transfer size of each BC and total sequential transfer sizes.

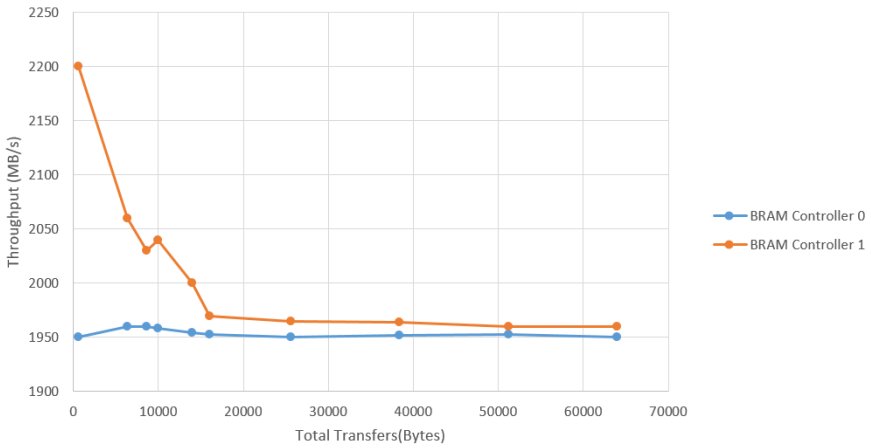


FIGURE 6.1: Performance of Master HP0, HP1 ports when data transfers are 640 bytes (for each BC).

system reached a steady point from the start. Below we present the metrics(latency and throughput) for the rest of the experiments. We used two memory attributes to make the BRAM cacheable, normal write-through and normal write-back. We saw an improvement when using these attributes (making the BRAM cacheable is also recommended by Xilinx to increase throughput). We flushed the memory and checked through the ILA core to make sure that the write did happen.

Figure 6.4 presents the results of sequential data access with different transfer sizes. In the last case (2 MB per BC), we transferred a total of 400 MB to the BRAM. Since BRAM only has 2 MB of space, data were overwritten. In the future, when we obtain FPGAs with more BRAM space, we can repeat the calculations so that data are not overwritten to obtain more accurate throughput results.

Figure 6.5 depicts the average latency, the average time it took to transfer data from the PS to the PL. We can see that up to 20 KB of the burst size, the latency is relatively low (no more than 100 us). This is because all accesses are made in the L1 cache and not the actual DDR. However, after 20 KB, we see that latency increases because all accesses are made in the L2 shared cache (which has higher access latency). Finally, after 1 MB of the burst size, we see an acute

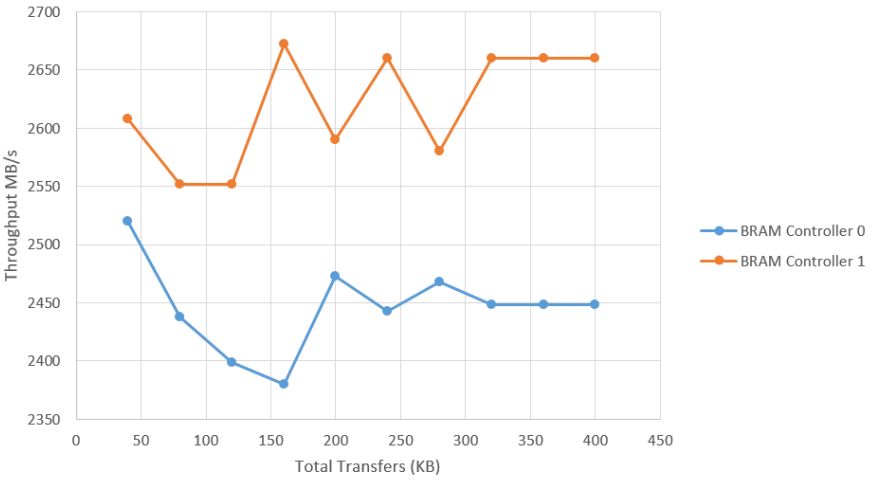


FIGURE 6.2: Performance of Master HP0, HP1 ports when data transfers are 2 KB (for each BC).

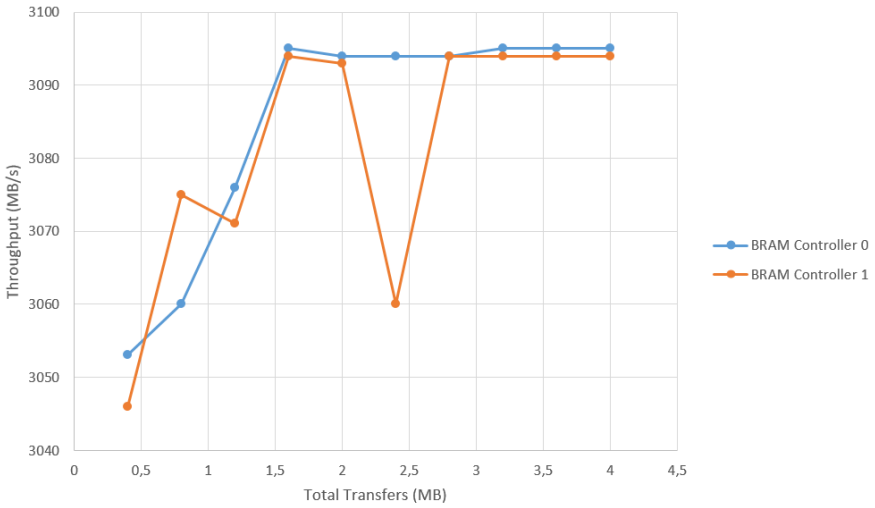


FIGURE 6.3: Performance of Master HP0, HP1 ports when data transfers are 20 KB (for each BC).

increase in latency; memory accesses are made in the actual DDR, not the cache. This means that the DDR controller does not load the cache only with the line buffers, as originally thought, but with 32 KB of data that reside in the following addresses from the one we requested.

The results provide a clear image of how the memory operates in sequential data accessing and how we should request data from the DDR. For sequential burst sizes of 32 KB and less (32 KB is the size of L1 cache), data are retrieved from the cache. Requesting data with cache enabled achieves higher throughput and less latency by prefetching data since the access pattern is sequential. Increasing the requested amount of data per burst or disabling the cache can lead to actual DDR accesses, drastically decreasing throughput and increased latency. When requesting sequentially stored data, we should aim for cache prefetching, with burst sizes up to 32 KB or in worst cases 1 MB. We also experimented with write

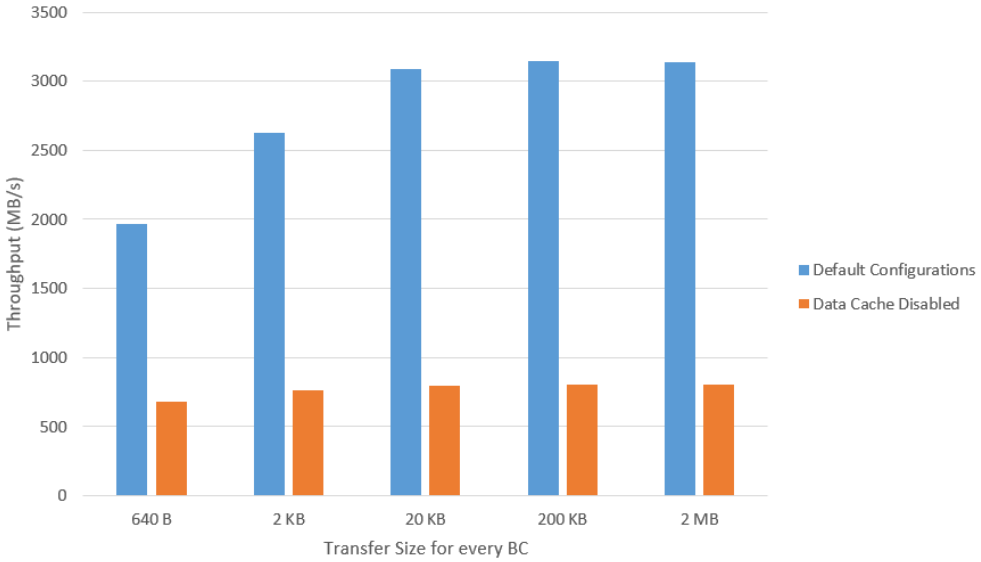


FIGURE 6.4: Results for burst access in sequentially stored data.

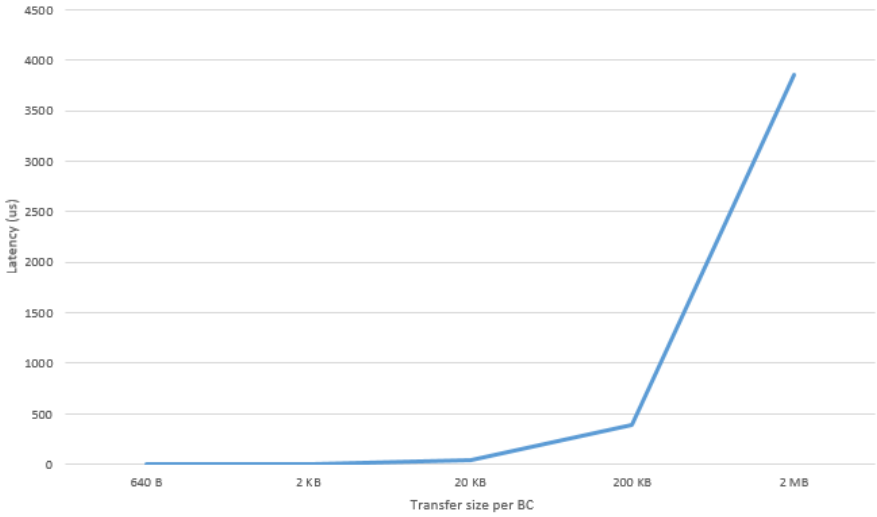


FIGURE 6.5: Latency results for various burst sizes per BC.

transactions from PL to PS, confirming the results presented in [51]. Writing (to the DDR) with cache disabled is faster than writing with cache enabled since, in the first case, the system does not wait for the data to be committed to the memory.

Table 6.6 depicts the average clock cycles per BC for different burst sizes. We see that clock cycles are on par with the throughput results. BC1 appears to be completing transactions faster than BC0. Moreover, we see that as we increase each transaction's burst size, the average clock cycles drop until they reach a saturation point of 3.1 clock cycles per word.

Transfer Size (in words)	BC0	BC1
80	5.2	4.77
250	3.92	3.82
2500	3.2	2.87
25000	3.12	3.12
250000	3.11	3.11

TABLE 6.6: Average clock cycles per BC per word for different word size bursts

6.2.2 Results for random data accessing

In this set of experiments, we read data that reside in non-consecutive addresses. Starting from a base address, we requested 640 bytes of data from the first BC. Then we moved 1 KB of data addresses and requested another 640 bytes with the second BC. This is done to simulate multiple PEs that request data from the main memory at different times and in different addresses. Since in real-world applications, a memory request is usually followed by many data that reside in consecutive addresses, we do not measure latency and throughput for a single word but many words. This random address request experiment involves random access at an address and sequential accesses at the following addresses. This is done to fully simulate a demanding hardware accelerator that requires data at specific memory addresses to start computations in different dimensions. We can understand that if we utilize more ports that address the DDR for data, then the overall memory subsystem will require more transactions to be completed faster.

Transfer Size for each BC	Number of iterations	Stride between BC access
640 Bytes	100	1 KB, 2KB, 20 KB, 200 KB, 2MB
2 KB	100	1 KB, 2KB, 20 KB, 200 KB, 2MB
20 KB	100	1 KB, 2KB, 20 KB, 200 KB, 2MB
200 KB	100	1 KB, 2KB, 20 KB, 200 KB, 2MB
2 MB	50	1 KB, 2KB, 20 KB, 200 KB, 2MB

TABLE 6.7: Transfer size of each BC and stride distance per BC.

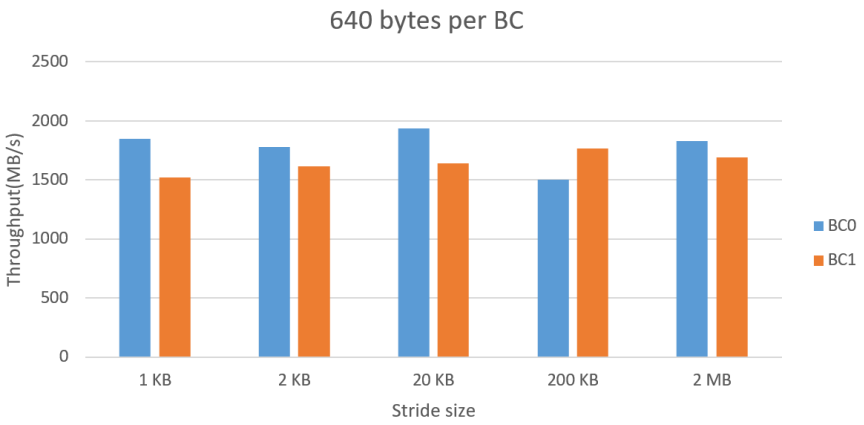


FIGURE 6.6: Results for different memory strides when each BC requested 640 Bytes.

In the images 6.6 to 6.10 we present the results for different data transfer sizes and different stride sizes. Table 6.8 summarizes the throughput differences between the two controllers based on the stride size. As we can see, random access has a huge impact on the second BC who requests data that are far from the data requested from the BC0. As the stride increases, the throughput gap between the two BCs increases as well. We can see that for request sizes smaller than the page size (2 KB), the gap is smaller than for bigger request sizes. Stride distance also plays an important role since as we increased the stride distance, the difference in throughput between the two BCs increased. This happens because data no longer

Stride distance	Data / BC				
	640 Bytes	2 KB	20 KB	200 KB	2 MB
1 KB	322 MB/s	164.6 MB/s	298.34 MB/s	-269.55 MB/s	136.92 MB/s
2 KB	453,57 MB/s	585,01 MB/s	450,93 MB/s	524,63 MB/s	514,27 MB/s
20 KB	200,18 MB/s	699,18 MB/s	1050,13 MB/s	1049,81 MB/s	969,92 MB/s
200 KB	441,02 MB/s	485,74 MB/s	1311,63 MB/s	1284,1 MB/s	1433,94 MB/s
2 MB	1197,4 MB/s	1325,83 MB/s	1427,77 MB/s	1425,51 MB/s	1421,44 MB/s

TABLE 6.8: Difference in in MB/s between the two BCs.

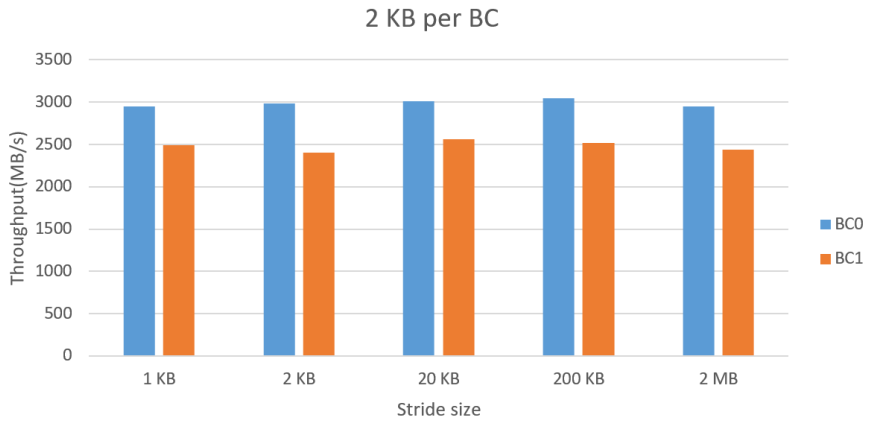


FIGURE 6.7: Results for different memory strides when each BC requested 2 KB.

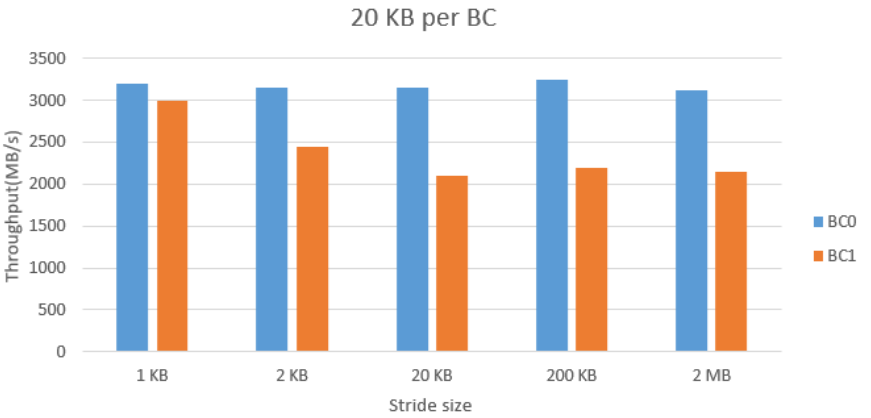


FIGURE 6.8: Results for different memory strides when each BC requested 20 KB.

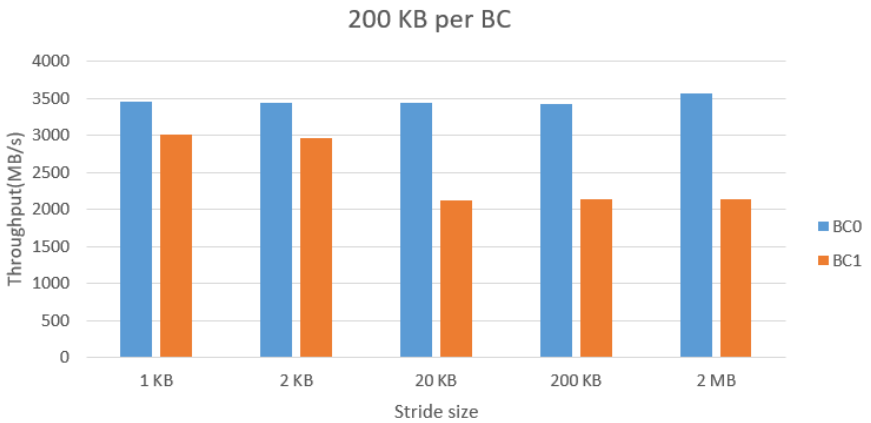


FIGURE 6.9: Results for different memory strides when each BC requested 200 KB.

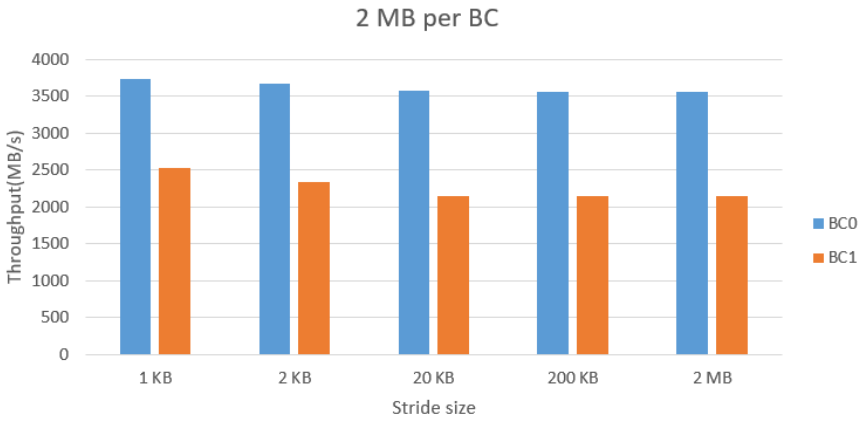


FIGURE 6.10: Results for different memory strides when each BC requested 2 MB.

Transfer Size	**640 Bytes**		**2 KB**		**20 KB**		**200 KB**		**2 MB**	
Stride	BC0	BC1	BC0	BC1	BC0	BC1	BC0	BC1	BC0	BC1
1 KB	5.2	6.3	3.25	3.84	2.31	3.16	2.15	3.06	2.11	3.06
2 KB	5.4	5.95	4	7.2	2.35	3.12	2.16	3.05	2.1	3.05
20 KB	4.95	5.85	3.18	3.74	2.32	3.11	2.16	3.06	2.1	3.05
200 KB	6.4	5.4	3.15	3.8	2.31	3.11	2.16	3.05	2.1	3.05
2 MB	5.6	10.9	3.25	3.94	2.34	3.11	2.1	3.06	2.1	3.05

TABLE 6.9: Average Clock Cycles per word for various transfer sizes (640 B, 2KB, 20 KB, 200 KB, 2 MB) and strides between BCs

resides in the cache, and the DDR controller must fetch them from the DDR itself. The negative value represents a particular case where transfer from BC1 achieved higher throughput. However, this was a local peak and is not does not represent how memory responds.

From the tables 6.8, 6.9 and the charts, we conclude that we must request a larger amount of data each time we access the DDR in random accesses. Accessing the DDR and requesting only 20 KB while accessing again but 2 MB of address further lowers the second BC performance by 1.5 GB/s. This is the same as requesting 2 MB of data with BC0, having a stride of 2 MB, and requesting 2MB with the second BC. Both cases have the same drop in performance between the requests, but we have transferred 100 times more data in the second case. However, this can be a problem for this particular FPGA since BRAM space is only 4 MBs, and we can not store 2 MB of data in the BRAM per memory request without having data overwritten or extremely fast accelerators. We believe that in modern FPGAs, with more memory resources, this will not be a problem. Furthermore, bursting more significant amounts of data increases throughput and lowers the required clock cycles per word, as shown in table 6.9.

6.2.3 Summarizing and Comparing with the Theoretical Results

In order to understand how the controller and the memory behave in various requests, we performed two types of experiments. With the sequential data access, we simulated 1-D applications where strides in memory are not required, and data can be obtained using single bursts. To fully optimize the design, the use of DMAs is necessary. Albeit, in multi-dimensional applications, DMAs are useful in only one dimension, and memory access is achieved in strides hence lowering the system's performance. As we mentioned before, the maximum theoretical **DDR4 bandwidth is 19.2 GB/s** whereas the maximum **AXI4 bandwidth per port is 4.8 GB/s**. Table 6.10 summarizes the results of our experiments, comparing them with the theoretical ones.

Sequential Access		**Random Access**				
		Stride Lenght				
Size per BC		1 KB	2KB	20 KB	200 KB	2 MB
640 B	1954.55	1845.96	1777.6	1932.2	1499.85	1828.38
2 KB	2630.12	2948.1	2984.77	3014.77	3045.38	2948.1
20 KB	3079.39	3200.85	3150.14	3150.25	3250.26	3120.12
200 KB	3141.57	3456.71	3441.86	3438.57	3425.8	3564.38
2 MB	3140.8	3728.48	3666.81	3567.96	3565.71	3562.69

Table 6.10: Throughput in MB/s for sequential and random access for various burst sizes and strides.

From the table above, we can see the vast impact the AXI4 port's bandwidth has on the system's performance. Even though the DDR4 module can support up to 19.2 GB/s in memory bandwidth, the AXI4 bandwidth per port is 4.8 GB/s. Even the best achievable throughput (per port) of 3.14 GB/s in sequential access is 34.55 % decreased when compared with the maximum AXI4 port throughput. A 6.28 GB/s can be achieved for two ports, providing a 67.29 % decrease compared to the DDR4 maximum available throughput. For random access, the maximum achievable throughput per port was 3.77 GB/s, which is 21.45 % smaller than the maximum AXI4 throughput, and for two ports, a 7.54 GB/s throughput can be achieved which is 60.72 % decreased when compared with the maximum DDR4 throughput. Utilizing more ports (for example, the four AXI slave ports of ZYNQ) produces 10.2 GB/s throughput, which is still 46.87 % smaller than the DDR4 maximum throughput when the memory module is running at a 1200 MHz clock.

Considering the information mentioned above and the tables 3.1 and 6.9, we can estimate the system's performance for an application. Given a representation of single-precision floating-point and no parallelism at any level (meaning that the next layer will start executing as soon as the previous layer completes the last

calculation), the Conv3-64 layer of VGGNet16 requires 7.1 KB of data or 1775 words.

Since we do not have parallelism, the data required for this layer is stored in sequential memory locations and accessed by the first BC. This means that for a burst of 7.1 KB, the total amount of clock cycles required to fetch the data is between 2.31 and 3.25 clock cycles per word, meaning that for 1775 words we need 4100.25 - 5768.75 clock cycles. However, this concerns only the network's parameters and not the input data. The second BC will need to make a stride of 553.69 MB (since VGGNet16 has $138,423,208$ $parameters \times 4$ $Bytes$ $each$) to access the input data. As we mentioned above, after a certain threshold (most of the time between the mark of 2-5 MB), the memory access time is the same, meaning that for a stride of 20 MB or a stride of 500 MB, the memory controller will have the same response (in clock cycles). For an input image of $224 \times 224 \times 3 = 602.1$ KB/image, the second BC will require 3.05 clock cycles (because the second BC will make a stride of 553.69 MB and fetch an amount between 200 KB and 2 MB of data). This means that to calculate all the outputs from the first convolutional layer of VGGNet16, we will need 5.36 - 6.03 clock cycles per word. It is worth mentioning that for this example, we do not have memory considerations meaning that both the input image and the layer can fit in the BRAM available, which is true since 7.1 KB of $parameters + 602.1$ KB for the $image < 4$ MB of $available$ $BRAM$. However, in real-world applications, this will not be the case since the designs most of the time are pipelined, and we need to fit multiple layers, their outputs, and the parameters in the BRAM available, meaning that we will need to do multiple smaller bursts.

The next layer, Conv3064 [3] requires 147.712 KB of data or 36928 words. If we assume that the design is pipelined, then the second BC can start fetching small amounts of data from the DDR4 as soon as the first few calculations of Conv3-64 are completed. The second BC will need a memory stride of 7 KB to access the parameters for the second layer. Assuming that our design does not fit in the FPGA (which is the most usual case), we can access 1-2 KB of data for the first layer, stride 7 KB, and access another 1-2 KB for the second layer. This means that to produce the first results of the first two layers of VGGNet16, we will need 3.18 - 7.2 clock cycles per word (since the stride size is between 2 and 20 KB).

6.3 Unexpected Port Bandwidth

As we mentioned in 5.5, we used memory attributes as described in the Memory Management Unit (MMU) file in Vivado SKD. These attributes change the

[3] https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96

cacheability and shareability of a specified memory region according to our needs. During the sequential data access experiment, we used two of these attributes making the BRAM space we were going to write the data cacheable. The memory in that region is outer shareable but not cacheable.

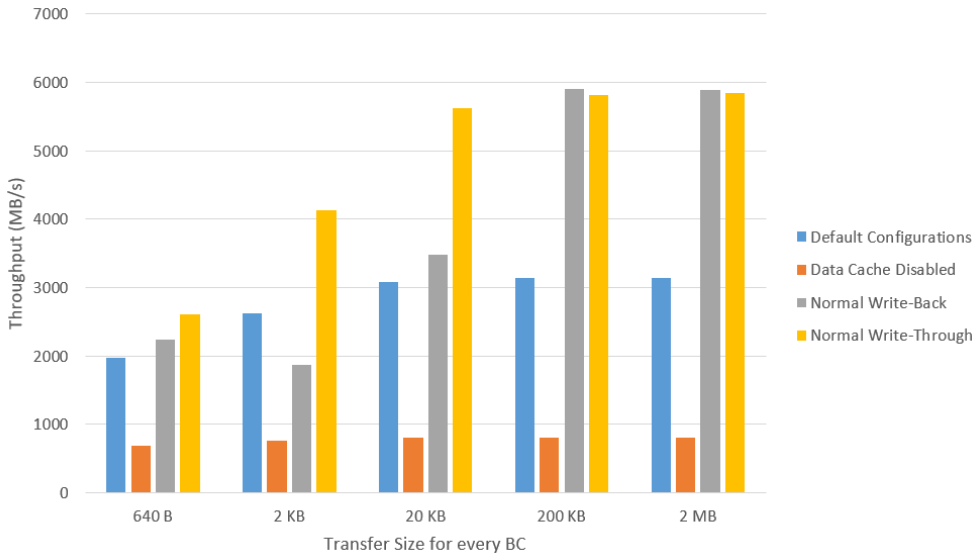


FIGURE 6.11: Measured throughput using two memory attributes.

In figure 6.11 we present the system's throughput by using default configurations (meaning that data cache is enabled and BRAM is non-cacheable) and with the data cache disabled. As was expected, the performance drops since all accesses hit the physical address of the DDR and not the cache lines. However, when we set BRAM cacheable (normal write-back and normal write-through cacheable and outer shareable), we noticed that the throughput exceeded the upper limit of 4.8 GB/s. To be more specific, for 640 bytes burst per BC, we notice a 14.51 % increase in performance when using the write-back attribute and a 33.44 % increase when using the write-through memory attribute. However, as we increased the transfer size to 20 KB, 200 KB, and 2 MB, we noticed that throughput soared, surpassing the upper throughput limit of 4.8 GB/s set by the AXI ports. Of course, this can not be achieved, and we are not sure if the controller does not consider some accesses providing us with false calculations since we do not have access to Xilinx's IP cores. This example depicts how even the results of an experimental calculation can be questioned, and the designer can not rely on these outcomes. We have discussed these findings with designers from FORTH, confirming their questionability, but we did not manage to conclude with a possible explanation. Furthermore, we have created a post in Xilinx's forums [4], describing the experiment and our findings, hoping to elaborate on this unusual behavior.

[4] https://forums.xilinx.com/t5/Other-FPGA-Architecture/Make-BRAM-cacheable/m-p/1215637

Chapter 7

Conclusions and Future Work

In this chapter, we will sum up and evaluate the information presented and the work that has been completed in this thesis. We will also discuss possible opportunities that might arise and how this thesis can be used and expanded in the future. We will also present extensions and optimizations.

7.1 Conclusions

FPGAs are frequently used in Convolutional Neural Networks since they provide hardware acceleration and vast configuration abilities while maintaining high power efficiency. Memory is the limiting factor of FPGAs' extensive use, and the memory subsystem and data obtain mechanism requires further research. This study used the ZCU102 FPGA, a Zynq UltraScale+^{TM} MPSoC with a quad-core Arm Cortex-A53. Even though modern FPGAs consist of dozens of GB in DDR memory (which is relatively slow), the small amount of memory in the Programmable Logic (which is a few MB and very fast) requires excellent handling of memory resources when creating hardware accelerators. Whether we want to implement applications that consist of 1-D or multi-dimensional computations, we must know the memory system's limits and exploit them.

Knowing how the external memory module of any given FPGA system is mapped, we can configure our system based on the application's needs. Reorganizing the address bit registers to minimize row access provides better performance results, depending on each application. Bank switching is better than row switching. Hence, putting the bank group bits at the LSBs of the memory address hides RAS since the DDR4 controller can be opening or closing a row in one bank group while accessing another bank group. Memory management attributes are required in order to speed our design. Based on the frequency of accesses in memory space, we can configure the memory to be cacheable or not and even choose the shareability across different PEs. Memory attributes can configure the PL BRAM, and changing the main memory configurations does not optimize the design. Furthermore, BRAM is very fast and can be cascaded according to our needs. BRAM

utilization is not optimal by the tools and is something that requires further optimization. Moreover, in a design with modules that access the same external memory subsystem, the second module will experience an increase in throughput when transferring data to the PL if the data are sequentially stored. In sequential accesses, burst sizes smaller than the L1 cache provide the most optimal latency solution, whereas bursts larger than the L2 cache access the DDR directly, and latency drastically increases. In random access, bursts should request large amounts of data, especially if the stride in memory is significant (something that happens in multidimensional applications with many filters and layers). However, this can be a problem in FPGAs where on-chip memory resources are limited.

7.2 Future Work

This work provides an insightful evaluation of a memory subsystem and the ways we can exploit the hardware capabilities to their maximum. However, there are some improvements and suggestions to further optimize memory transfers in any given memory subsystem:

- Connecting the PL BRAM through AXI BRAM Controller is not optimal when speed is essential. To further speed the design, we need to create a custom AXI slave controller that exploits the memory capabilities.
- The need to further explore problems that cause idle states, data collisions, or memory misses is vital to fully understand how the memory subsystem of a given architecture operates.
- Furthermore, future work can research the way data are transferred from the PL to the PS, efficient ways to store data in the DDR, and even explore the possibilities of an efficient memory controller that writes data from the PL to the PS.
- By exploiting the results provided in this thesis, one can design an optimized and accurate application or algorithm. This thesis results provide guidelines on how to retrieve data from memory system efficiently for future designs.
- Since the QFDB features 4 ZCU102 FPGAs and many capabilities, the next step is to evaluate this board's memory subsystem. Since data are stored in different DIMMS, it would be very beneficial to measure the memory's abilities and study how data are stored and retrieved.

References

[1] D. Buscombe and A. C. Ritchie, "Landscape Classification with Deep Neural Networks", *Geosciences*, vol. 8, 2018. [Online]. Available: https://www.mdpi.com/2076-3263/8/7/244/htm.

[2] D. Silver, A. Huang, C. Maddison, *et al.*, "Mastering the game of Go with deep neural networks and tree search", *Nature*, vol. 529, 484–489, 2016. [Online]. Available: https://www.nature.com/articles/nature16961#article-info.

[4] F. Chaix, A. Ioannou, N. Kossifidis, N. Dimou, G. Ieronymakis, M. Marazakis, V. Papaefstathiou, *et al.*, "Implementation and Impact of an Ultra-Compact Multi-FPGA Board for Large System Prototyping", *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pp. 34–41, DOI: 10.1109/H2RC49586.2019.00010.

[5] G. Pitsis, G. Tsagkatakis, C. Kozanitis, I. Kalomoiris, *et al.*, "Efficient convolutional neural network weight compression for space data classification on multi-fpga platforms", *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3917–3921, 2019. DOI: 10.1109/ICASSP.2019.8682732.

[7] J. Singh Yadav and M. Jain, "Cache Memory Optimization", 6, vol. 1, 2013, pp. 119–125. [Online]. Available: https://pdfs.semanticscholar.org/22bd/665508754ac67e608447ef010e37c06c3d0c.pdf.

[8] V. Chaplot, "Cache Memory: An Analysis on Performance Issues", *International Journal of Advance Research in Computer Science and Management Studies*, vol. 4, no. 7, pp. 26–28, 2016. [Online]. Available: http://www.ijarcsms.com/docs/paper/volume4/issue7/V4I7-0014.pdf.

[9] Q. Javaid, A. Zafar, M. Awais, and M. A. Shah, "Cache Memory: An Analysis on Replacement Algorithms and Optimization Techniques", vol. 36, Mehran University of Engineering and Technology, Jamshoro, Pakistan, Oct. 2017, pp. 831–840. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01700364.

[10] K. Patidar and R. Gupta, "A Taxonomy of Cache Replacement Algorithms", *International Journal of New Technologies in Science and Engineering*, vol. 2, no. 3, Sep. 2015. [Online]. Available: https://ijntse.com/upload/1443770072IJNTSE-SP-125.pdf.

[11] M. Abd-El-Barr and H. El-Rewini, *Fundamentals of computer organization and architecture*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2005, pp. 107–108.

[12] R. Melo, D. Caruso, and S. Tropea, "Memory-mapped I/O over dual port BRAM on FPGA", pp. 1–6, Mar. 2012. DOI: 10.1109/SPL.2012.6211780.

[13] L. H. Crockett, D. Northcote, C. Ramsay, F. D. Robinson, and R. W. Stewart, *Exploring Zynq® MPSoC With PYNQ and Machine Learning Applications*. Strathclyde Academic Media, 2019, p. 614, ISBN: 978-0-9929787-5-4. [Online]. Available: https://www.zynq-mpsoc-book.com/wp-content/uploads/2019/04/MPSoC_ebook_web_v1.0.pdf.

[14] D. A. Ioannou, "UniLogic (Unified Logic):A Scalable Architecture for Increased Programmability in Highly Parallel ReconfigurableSystems", 2020. [Online]. Available: https://dias.library.tuc.gr/view/84933?locale=el.

[15] J. G. Pandey, A. Karmakar, C. Shekhar, and S. Gurunarayanan, "An FPGA-Based Architecture for Local Similarity Measure for Image/Video Processing Applications", *2015 28th International Conference on VLSI Design*, pp. 339–344, 2015. DOI: 10.1109/VLSID.2015.63.

[16] C. Torres-Huitzil and M. A. Nuño Maganda, "Areatime Efficient Implementation of Local Adaptive Image Thresholding in Reconfigurable Hardware", *SIGARCH Comput. Archit. News*, vol. 42, no. 4, 33–38, Dec. 2014, ISSN: 0163-5964. DOI: 10.1145/2693714.2693721.

[17] J. Y. Mori, F. Kautz, and M. Hübner, "Efficient Camera Input System and Memory Partition for a Vision Soft-Processor", in *Applied Reconfigurable Computing - 12th International Symposium, ARC 2016, Mangaratiba, RJ, Brazil, March 22-24, 2016, Proceedings*, vol. 9625, Springer, 2016, pp. 328–333. DOI: 10.1007/978-3-319-30481-6_27.

[18] R. Chen, N. Park, and V. K. Prasanna, "High throughput energy efficient parallel FFT architecture on FPGAs", in *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*, IEEE, 2013, pp. 1–6. DOI: 10.1109/HPEC.2013.6670343.

[19] H. Sahlbach, R. Ernst, S. Wonneberger, and T. Graf, "Exploration of FPGA-based dense block matching for motion estimation and stereo vision on a single chip", in *2013 IEEE Intelligent Vehicles Symposium (IV), Gold Coast City, Australia, June 23-26, 2013*, IEEE, 2013, pp. 823–828. DOI: 10.1109/IVS.2013.6629568.

[20] C. H. Chou, A. Severance, A. D. Brant, *et al.*, "VEGAS: soft vector processor with scratchpad memory", in *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, ACM, 2011, pp. 15–24. DOI: 10.1145/1950413.1950420.

[21] P. Garcia, D. Bhowmik, R. Stewart, G. Michaelson, and A. Wallace, "Optimized Memory Allocation and Power Minimization for FPGA-Based Image Processing", *Journal of Imaging*, vol. 5, no. 1, p. 7, 2019, ISSN: 2313-433X. DOI: 10.3390/jimaging5010007.

[22] G. Dessouky, M. Klaiber, D. G. Bailey, and S. Simon, "Adaptive Dynamic On-chip Memory Management for FPGA-based reconfigurable architectures", in *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, IEEE, 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927471.

[23] M. Klaiber, D. G. Bailey, S. Ahmed, Y. Baroud, and S. Simon, "A high-throughput FPGA architecture for parallel connected components analysis based on label reuse", in *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*, IEEE, 2013, pp. 302–305. DOI: 10.1109/FPT.2013.6718372.

[24] M. Supriya and N. R. Kumar, "BRAM-based Multiported Memory Designs on FPGA", in *International Journal of Advanced Research in Computer and Communication Engineering, November 2017*, vol. 6, IJARCCE, 2017, pp. 1–5.

[25] J. Lin and B. C. Lai, "BRAM efficient multi-ported memory on FPGA", in *VLSI Design, Automation and Test, VLSI-DAT 2015, Hsinchu, Taiwan, April 27-29, 2015*, IEEE, 2015, pp. 1–4. DOI: 10.1109/VLSI-DAT.2015.7114526.

[26] S. N. Shahrouzi, A. Alkamil, and D. G. Perera, "Towards composing optimized bi-directional multi-ported memories for next-generation fpgas", *IEEE Access*, vol. 8, pp. 91 531–91 545, 2020. DOI: 10.1109/ACCESS.2020.2994882.

[27] K. Kara, J. Giceva, and G. Alonso, "FPGA-based Data Partitioning", in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, ACM, 2017, pp. 433–445. DOI: 10.1145/3035918.3035946.

[28] N. Kerkiz, A. Elchouemi, and D. Bouldin, "Multi-FPGA Partitioning Method Based on Topological Levelization", *Journal of Electrical and Computer Engineering*, vol. 2010, Apr. 2010. DOI: 10.1155/2010/709487.

[29] P. Li, Y. Wang, P. Zhang, *et al.*, "Memory partitioning and scheduling co-optimization in behavioral synthesis", in *2012 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2012, San Jose, CA, USA, November 5-8, 2012*, A. J. Hu, Ed., ACM, 2012, pp. 488–495. DOI: 10.1145/2429384.2429484.

[30] Y. Ben Asher and N. Rotem, "Automatic Memory Partitioning: Increasing memory parallelism via data structure partitioning", in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010, pp. 155–161. DOI: 10.1145/1878961.1878989.

[31] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms", in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64. DOI: 10.1109/RTAS.2013.6531079.

[32] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis", May 2013, pp. 1–8. DOI: 10.1145/2463209.2488748.

[33] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth", *IEEE Trans. Parallel Distributed Syst.*, vol. 25, no. 3, pp. 695–705, 2014. DOI: 10.1109/TPDS.2013.51.

[34] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors", *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, 2009. DOI: 10.1137/070693199.

[35] K. Kalaitzis, E. Sotiriadis, I. Papaefstathiou, and A. Dollas, "Evaluation of External Memory Access Performance on a High-End FPGA Hybrid Computer", *Comput.*, vol. 4, no. 4, p. 41, 2016. DOI: 10.3390/computation4040041.

[36] A. G. Pitsis, "Design and implementation of an FPGA-based convolutional neural network accelerator ", 2018. [Online]. Available: https://dias.library.tuc.gr/view/79092?.

[37] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", vol. 25, Jan. 2012. DOI: 10.1145/3065386.

[38] K. Simonyan and A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014. arXiv: 1409.1556 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1409.1556.

[39] K. He, X. Zhang, S. Ren, and J. Sun, *Deep Residual Learning for Image Recognition*, 2015. arXiv: 1512.03385 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1512.03385.

[40] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going Deeper with Convolutions", *CoRR*, vol. abs/1409.4842, 2014. arXiv: 1409.4842. [Online]. Available: http://arxiv.org/abs/1409.4842.

[49] T. Fotakis, "Analysis and Design Methodology of Convolutional Neural Networks mapping on Reconfigurable Logic", 2020. [Online]. Available: https://dias.library.tuc.gr/view/86843.

[50] K. Manev, A. Vaishnav, and D. Koch, "Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems", in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 179–187. DOI: 10.1109/ICFPT47387.2019.00029.

[51] A. Bansal, R. Tabish, G. Gracioli, *et al.*, "Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC", Jul. 2018. [Online]. Available: http://cs-people.bu.edu/rmancuso/files/papers/mpsoc_mem_OSPERT18.pdf.

External Links

[3] "Zynq ultrascale+ mpsoc zcu102 evaluation kit", [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[6] "Vivado design suite - hlx editions", [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[41] *Vivado design suite user guide - design analysis and closure techniques*, English, version 2018.2, Nov. 2011, 308 pp. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug906-vivado-design-analysis.pdf.

[42] *Vivado design suite user guide - high-level synthesis*, English, version 2020.1, Jun. 2020, 589 pp. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf.

[43] *AXI Reference Guide - UG1037*, English, version 4.0, Jul. 2017, 175 pp. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[44] Micron, "DDR4 SDRAM SODIMM MTA4ATF51264HZ – 4GB", English, [Online]. Available: https://gr.mouser.com/datasheet/2/671/atf4c512x64hz-1284544.pdf.

[45] *AXI Block RAM (BRAM) Controller - PG078*, English, version 4.0, Oct. 2016, 85 pp. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_0/pg078-axi-bram-ctrl.pdf.

[46] *AXI SmartConnect - PG247*, English, version 1.0, Feb. 2020, 55 pp. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/smartconnect/v1_0/pg247-smartconnect.pdf.

[47] *Block Memory Generator - PG058*, English, version 8.4, Dec. 2019, 129 pp. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf.

[48] Xilinx, *Ultrascale architecture memory resources*, English, version 1.11, Aug. 2020, 138 pp. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.

[52] *UltraScale ArchitectureLibraries Guide - UG974*, English, version 2020.2, Dec. 2020, 697 pp. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_0/pg078-axi-bram-ctrl.pdf.